FM-QoS: Real-time Communication using Self-synchronizing Schedules

Kay Connelly and Andrew A. Chien
{hane, achien}@cs.uiuc.edu

Concurrent Systems Architecture Group
Department of Computer Science, University of Illinois
1304 W. Springfield Ave.
Urbana, IL 61801
`http://www-csag.cs.uiuc.edu`

Abstract

FM-QoS employs a novel communication architecture based on network feedback to provide predictable communication performance (e.g. deterministic latencies and guaranteed bandwidths) for high speed cluster interconnects. Network feedback is combined with self-synchronizing communication schedules to achieve synchrony in the network interfaces (NIs). Based on this synchrony, the network can be scheduled to provide predictable performance without special network QoS hardware. We describe the key element of the FM-QoS approach, feedback-based synchronization (FBS), which exploits network feedback to synchronize senders. We use Petri nets to characterize the set of self-synchronizing communication schedules for which FBS is effective and to describe the resulting synchronization overhead as a function of the clock drift across the network nodes. Analytic modeling suggests that for clocks of quality 300 ppm (such as found in the Myrinet NI), a synchronization overhead less than 1% of the total communication traffic is achievable -- significantly better than previous software-based schemes and comparable to hardware-intensive approaches such as virtual circuits (e.g. ATM).

We have built a prototype of FBS for Myricom's Myrinet network (a 1.28 Gbps cluster network) which demonstrates the viability of the approach by sharing network resources with predictable performance. The prototype, which implements the local node schedule in software, achieves predictable latencies of 23 μs for a single-switch, 8-node network and 2 KB packets. In comparison, the best-effort scheme achieves 104 μs for the same network without FBS. While this ratio of over four to one already demonstrates the viability of the approach, it includes nearly 10 μs of overhead due to the software implementation. For hardware implementations of local node scheduling, and for networks with cascaded switches, these ratios should be much larger factors.

1. Introduction

Clustered computers are an increasingly attractive approach to building scalable servers for supercomputing, database, and World-Wide Web applications. These systems consist of a number of desktop or 2 or 4-way symmetric multiprocessors connected by a high speed cluster interconnect. There has been a great deal of research in high performance communication layers for these platforms [1, 5, 25], resulting in communication layers such as Illinois Fast Messages (FM) which delivers minimum latencies of 8 μs, and peak bandwidths of ~76 Megabytes/second [4]. This performance is high enough to support many supercomputing, multimedia, and embedded applications. However to date, these communication layers do not provide guaranteed predictable performance, as network contention and scheduling effects can cause delivered performance to vary by several orders of magnitude. Embedded and multimedia applications typically require predictable performance [20, 9, 14], and recent studies have demonstrated that predictable performance can benefit supercomputing applications designed for best-effort networks [18]. Thus, our goal is to develop a network architecture which delivers both predictable, high performance for cluster interconnects, thereby supporting all three application classes in a single network environment.

While we are prototyping FM-QoS in the context of wormhole-routed switches from Myrinet, we believe our techniques are applicable to any network that provides link-level flow control feedback to the network interface (NI) – in hardware or software. Wormhole routers [6] are popular because of their low cost and high performance which derive from their simplicity. However, wormhole routing makes predictable performance difficult because the "stop in place" flow control protocol induces network blocking. The transitive effects of such blockage can be catastrophic for quality of service. An example of such packet blockage is shown in Figure 1. The packet from node 1 to node 8 spans switches A, B and C, while the packets from nodes 2, 4 and 7 are blocked, waiting on ports used by node 1's packet. Cascaded switches compound the problem, so the worst case delay grows rapidly with network size[1] [18,19].

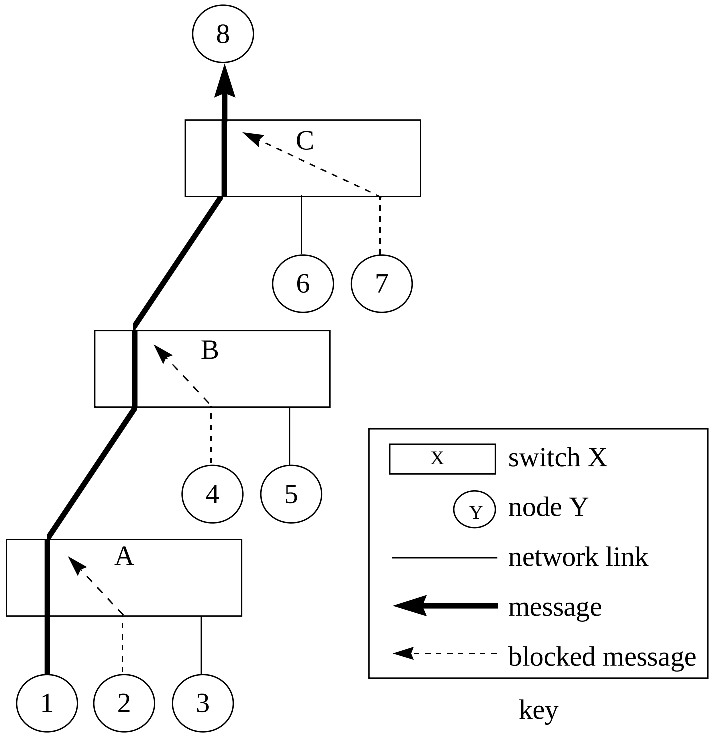


Figure 1: Blocking across switches in a wormhole-routed network can cause delays proportional to the number of switches the packet must traverse.

To achieve better quality of service guarantees, our approach, **feedback-based synchronization** (FBS), exploits network flow control information and self-synchronizing schedules to produce a global view of time. This global notion of time is then used to implement communication schedules which have no resource conflicts, resulting in a blockage-free network. Thus

[1] For 8 port switches, the worst-case time a packet may be in the network is $(7^n + 1)t$, where n is the number of switches that must be traversed and t is the time it takes for a packet to traverse the network.

communication delay consists of two distinct parts -- queueing for network entry (a small, bounded delay) and network traversal (a small, bounded delay determined by the number of links traversed).

1.1 Example

Network feedback and a self-synchronizing schedule are sufficient to maintain node synchronization. To demonstrate, we describe an example based on the communication schedule in Figure 2. The timelines in Figure 3 depict three attempts to implement the schedule. Figure 3a illustrates idealized synchronous execution; however in practice, nodes will experience some clock drift in their clocks, introducing a skew between the different nodes' execution of the schedule. Figure 3b demonstrates how such skew can result in faulty execution of the global schedule. In this case, Node B's clock is too slow, and thus its second message to C arrives too late, missing its window for QoS. Such slips can cascade, quickly producing a dynamic schedule that bears little resemblance to the ideal synchronized schedule.

	NODE A	NODE B
Time Slot 1	A → D	B → F
Time Slot 2	A → E	B → C
Time Slot 3	A → D	B → F
Time Slot 4	A → D	B → F
Time Slot 5	A → E	B → C
Time Slot 6	A → C	B→ F

Figure 2. Communication schedule for two nodes A and B sending to nodes C, D, E and F.

Feedback-based synchronization provides a mechanism to enforce global synchrony. By changing the global schedule (A's destination for slot 3), an extra dependence is introduced. This corrects the skew between nodes A and B, ensuring the schedule is executed correctly as shown in Figure 3c. By incorporating such dependences throughout a resource schedule, we can produce a self-synchronizing schedule where nodes will never lose synchrony. The frequency of dependences required depends upon the NI's maximum clock drift rate.

1.2 Feedback-Based Synchronization and Self-synchronizing Schedules

To avoid all blocking within the network, the communication schedule must be conflict-free (i.e. no two packets attempt to use the same network resources at the same time). Conflict-free schedules are easy to design, but challenging to implement without a global clock. The primary contribution of this paper is a scheme for synchronization based on network feedback. We define a class of **self-synchronizing schedules** which exploit network feedback to maintain global synchrony. In essence, communication traffic is added to a schedule in order to ensure that all nodes interact through the network. To characterize this class of schedules precisely, we use a Petri net model of network schedules, and define the Petri net structures which bound the allowable clock skew across nodes.

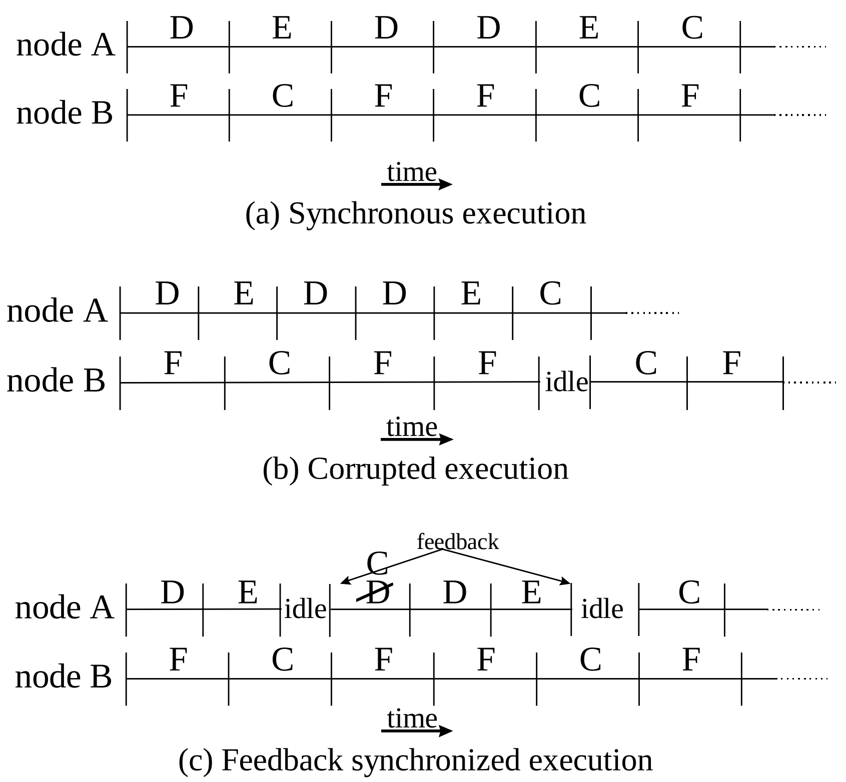


Figure 3. Time line of executions of the schedule in Figure 2. (a) Ideal, synchronous; (b) a corrupt schedule (node B is slower than node A); (c) a FBS dependence slows A, preventing schedule corruption.

Our design studies show that without any special hardware for quality of service (e.g. ATM's virtual circuits), a Myrinet-based implementation can provide guaranteed latencies on the order of tens of microseconds. We describe a prototype implementation based on Myricom's Myrinet network (a 1.28 Gbps cluster network) and FM. The prototype is implemented in device firmware on the network interface and incurs synchronization overheads less than 1% of the total communication traffic. For a single-switch, 8-node network, 2 KB packets can be sent between NIs with worst-case latencies of 23 μs, compared to 104 μs for the same network without any mechanisms for QoS. The software overhead incurred by our prototype is 10 μs, implying that a hardware implementation of the node scheduling could improve performance dramatically, from 4:1 to 8:1 for 8-port switches. In addition, FBS will improve worst-case latency bounds geometrically with respect to the number of switches a packet may traverse in a multi-switch network without QoS hardware.

1.3 Organization

The remainder of the paper is organized as follows. Section 2 describes the relevant background, surveying relevant details of Myricom's Myrinet and Illinois Fast Messages. Section 3 provides the high level view of the reservation and usage model. Section 4 describes the core of the approach, feedback-based synchronization and self-synchronizing schedules. In Section 5, we describe an analytical model and the measured performance of a FBS prototype. Section 6 discusses the results and related work. Finally, Section 7 summarizes the paper, and describes a range of possible research directions.

2. Background

The FM-QoS effort is part of both the Illinois High Performance Virtual Machines (HPVM) and End to end Performance vIa Quality of service (EPIQ) projects at the Department of Computer Science at the University of Illinois [4, 8]. The goal of the HPVM project is to support high performance distributed computing by building low-overhead, high performance communication software (e.g. FM [25] and MPI-FM [21]) and other middleware (coscheduling, resource management). The target deployment environments include high-performance embedded systems and scalable clusters. EPIQ's goal is to develop a framework for scalable, responsive and re-configurable end-to-end resource management. FM-QoS provides a high performance, controllable communication substrate for experimentation. Of course, network communication is just one of the many system resources which an EPIQ system must manage.

2.1. Myricom's Myrinet

Myrinet is a high speed local area network with full duplex, 1.28 Gbps links. Derived from multicomputer routers, current Myrinet offerings employ wormhole routing and 8x8 crossbar switches [2]. Arbitration in the crossbars is fair and done by a round robin algorithm. The network interface has a programmable CPU (the LANai) with 256KB of SRAM that attaches to the input/output bus of the host processor. For all experiments, this host is a 200Mhz Pentium Pro system, with 33Mhz, 32-bit PCI bus. The LANai is single issue, and runs at 33Mhz in our configuration. It also contains 3 DMA engines: host memory $\leftrightarrow$ LANai memory, LANai memory $\rightarrow$ network, and network $\rightarrow$ LANai memory. Thus, the LANai can send and receive from the network at the same time.

The wormhole routing flow control is coupled to the LANai's DMA, so the LANai can detect network feedback based on the latency of DMA operations. There is little buffering in the Myrinet switch, and messages sent over a particular path are delivered to the receiving LANai in order.

While many schemes are possible, in the Illinois Fast Messages system, the host and LANai work together to manage network communication. Their actions are coordinated through memory on the LANai which the host can map into its own address space, as well as through DMAing messages from the LANai's queue structures into comparable host queues. In our software prototype of FBS, the LANai is programmed to implement the NI local scheduling.

2.2 Illinois Fast Messages

Illinois Fast Messages (FM) is a low-overhead, high-performance software messaging layer with an implementation for the Myrinet network [25]. FM's primary goal is to provide high-performance not just to applications written directly to the FM API, but also to applications written to a wide range of higher-level communication APIs such as MPI, SHMEM Put/Get and Global Arrays [21, 4]. To implement these higher level API's efficiently, FM must provide the right set of delivery guarantees. Too weak or too strong will reduce performance. See [17, 25] for more detailed discussion of these issues.

FM provides the following guarantees to enable simple, high performance implementation of a wide range of user-level APIs:

- in-order delivery,
- reliable delivery, and
- decoupling of the host processor and the network.

Together, these guarantees isolate the management of underlying highly reliable, fast networks to the low-level substrate. Implementing these guarantees for FM on Myrinet is straightforward. The underlying Myrinet delivers packets in-order, so FM simply uses FIFO buffer to assure in-order delivery for applications. Reliable delivery is achieved by carefully managing the network so that packets are never stopped in place for long. This is achieved with a simple credit-based flow control mechanism. This avoids link resets (a mechanism for deadlock detection), allowing FM to treat the Myrinet network as reliable. Finally, FM avoids system calls into the operating system whenever possible, thereby minimizing OS overheads. Thus, instead of an interrupt-based scheme when a message arrives, the user receives messages through polling.

FM's interface is carefully designed to enable efficient composition into higher level layers. This composition eliminates copies and buffer pool overruns which often reduce performance significantly. The major concepts include:

- streams (efficient gather-scatter),
- receiver flow control, and
- receiver-side per-packet multithreading.

Streams allow users to specify source and destination for pieces of a message interleaved with arbitrary computation. This provides an easy scatter/gather mechanism for sending and receiving messages. Receiver flow control enables messaging layers to pace the rate at which data is removed from the network layer, avoiding buffer pool overruns and the resulting additional message copies. Finally, per-packet multithreading allows programmers to deal with each message as a logical unit, independent of hardware level packetization, simplifying the programming of user API's atop FM.

In FM v2.02, messages can be sent with overheads as low as 2.6 microseconds and with application-to-application latencies as small as 8 microseconds. Peak application-to-application bandwidths are 76 Megabytes per second (637 Mbps), greater than the 563 Mbps link speed of OC-12 ATM. Raw FM performance is competitive with other messaging layers written for Myrinet. The main difference is that FM's reliable, in-order delivery guarantees, combined with FM streams allows FM to provide more of the performance to protocols written on top of FM. For example, MPI-FM incurs overheads as low as 4 microseconds with latencies of 12 microseconds. MPI-FM's peak bandwidth is over 70 MB/s.

3. FM-QoS Overview

3.1 FM-QoS

FM-QoS is a real-time extension of the FM API. Extensions include calls to reserve and relinquish communication channels with fixed delay and bandwidth between network interfaces. Messages are sent and received against a reserved channel for guaranteed performance or simply delivered by best effort. Figure 4 lists the FM-QoS API. The `QoS` argument for FM_ChannelSetup and FM_ChannelNegotiate is a data structure that specifies the desired quality of service. This data structure contains typical QoS parameters such as minimum bandwidth, maximum latency, jitter constraints, etc. In this and following sections, we describe the implementation ideas behind FM-QoS.

QoS API extension	channel **FM_ChannelSetup**(destination, QoS)
	FM_ChannelDestroy(channel)
	FM_ChannelNegotiate(channel, QoS)

FM base API	stream **FM_BeginMessage**(channel, message_size, handler_id)
	FM_SendPiece(stream, buffer, buffer_length)
	FM_EndMessage(stream)
	FM_Extract(channel, length)

Figure 4: In addition to the send and receive functions found in the FM API, the FM-QoS API includes three functions for managing QoS channels.

It is important to note that FBS delivers predictable performance to the NI, not to the host application. This is because FBS does not ensure that the application will be scheduled on the host at the appropriate time; instead, our approach assumes the host application will be scheduled so that it can receive its packets in a timely manner. Other research in the HPVM project is addressing the issue of the host scheduling applications to respond to network events [4].

3.2 Global Resource Schedule

Our approach to quality of service depends on a global **resource schedule** which enforces predictable time-division multiplexing of network links and outputs. Because the schedule is enforced externally at the NIs, it achieves quality of service “end-to-end” without increasing switch complexity. For now, we assume fixed size packets and uniform link speeds. We call the resulting constant transmission time for each packet a **slot**. Thus, to ensure a particular quality of service, one need only reserve appropriate slots in the resource schedule. For example, in the cyclic resource schedule shown in Figure 5, the connection from node B to node A is ensured one-fourth the link bandwidth.

FM-QoS uses a reservation protocol [29] to determine the resource schedule. The communication needed to implement this protocol can be embedded in the synchronization packets necessary to ensure a self-synchronous schedule, eliminating any additional overhead. The reservation protocol produces self-synchronous schedules as defined in Section 4.

Enforcing a resource schedule is challenging because most LANs are asynchronous, i.e. there is no global clock. For example, to enforce the slot-schedule shown in Figure 5, the nodes must be synchronized in time at slot boundaries. Hardware approaches (e.g. [26]) are usually considered impractical, particularly for high speed, large scale networks. User level software schemes for clock synchronization or even software clocks are generally unable to provide the microsecond scale synchronization required for our slots (a few microseconds each).

	NODE A	NODE B	NODE C	NODE D
Time Slot 1	A → A			
Time Slot 2				D → A
Time Slot 3			C → A	
Time Slot 4		B→ A		

Figure 5. Cyclic communication schedule of period 4 for a 4-node crossbar. The connection from Node B to Node A is guaranteed one-fourth the link bandwidth.

3.3 Network Assumptions

FBS makes several assumptions about the network. FBS can be implemented on any network that satisfies these constraints.

NI Detects Network Feedback In the case of wormhole routed networks such as Myrinet, the NI detects packet blockage by delay in the completion of a DMA to the network. Thus, the NI can slow down its execution of the schedule by waiting for the DMA to finish before continuing on to the next slot.

Zero Packet Loss, In-order Delivery Packet loss and in-order delivery is a problem for networks such as ATM because they buffer entire cells on the switch and allow the switch to decide when to send or drop a cell due to network congestion. In networks such as Myrinet, the switch does not drop packets and an entire packet cannot be stored on the switch, easily satisfying these requirements.

NI Decoupled from Host Processor Having a dedicated, autonomous NI act as the network scheduler prevents scheduling problems on the host from interfering in the execution of the self-synchronous schedule.

4. Network Synchronization

In our approach to network synchronization, feedback-based synchronization (FBS), a node's notion of time is regulated by its interaction with other nodes via network feedback. As the example in Section 1 demonstrated, for FBS to work, there must be adequate interaction between network nodes. By introducing dependences into the communication schedule, we can ensure that all nodes' execution of the schedule will be governed by the slowest node. A primary task, then, is to determine how many dependences must be introduced: too few will result in a violated schedule while too many will over-restrict the communication and incur unnecessary overhead.

To reason clearly about the properties of schedules, we introduce a notation for schedule graphs based on Petri nets. The schedule formulation in Petri nets provides a way to cleanly separate the local and global dependences in a resource schedule; where the local dependences can be implemented as a finite state machines on the NIs, and the global dependences are enforced through network feedback. This partition allows us to define a class of schedules that can enforce a global notion of time based on the global dependences.

Further, by using the lengths of the paths in a Petri net, we can determine a bound on the clock skew between network nodes before the schedule may become corrupted. Conversely, given a particular clock drift rate for network nodes, we can place a bound on the path lengths in the Petri net that will ensure a correct execution of the communication schedule.

4.1 Schedule Graphs

Schedule graphs are Signal Transition Graphs (STG, Petri nets with labeled transitions) [12]. A Petri net is a 4-tuple PN = (P, T, E, μ^o). P is a finite set of places (graphically represented as circles). T is a finite set of transitions, representing events in the system (drawn as horizontal bars). E is a finite set of arcs such that $\{\forall e \; \varepsilon \; E \mid e \; \varepsilon \; PxT \text{ or } e \; \varepsilon \; TxP\}$, that is, the connections between places and transitions. μ^o is the initial marking of PN (initial token locations). Places can be empty or contain one or more tokens. When a place contains a token, it is said to be **active**. The set of all active places makes up the current state of the system. A transition is enabled when all of the places connected to its incoming arcs are active. A transition can fire only when it is enabled. When a transition fires, it removes one token from each of its inputs and places one token in each of its outputs.

To represent a schedule in a schedule graph, we first generate two transitions for each send slot in the schedule. These are labeled with positive and negative transitions for the source and destination pair, representing the starting and finishing times of a packet send, respectively. For example, AB^+ represents the start of a send from node A to node B; whereas, AB^- represents the end of that packet's same send. Each positive transition is linked by an arc to its corresponding negative transition. From each negative transition, two outgoing arcs are added, one to the sender's next send (positive transition) and the other to the closest following send (in time) to the same destination. Finally, a place is generated in the middle of each arc. For example, the schedule shown in Figure 6 (a superset of the schedule in Figure 5), produces a schedule graph as shown in Figure 7.

	NODE A	NODE B	NODE C	NODE D
Time Slot 1	A → A	B → B	C → C	D → D
Time Slot 2	A → B	B → C	C → D	D → A
Time Slot 3	A → C	B → D	C → A	D → B
Time Slot 4	A → D	B → A	C → B	D → C

Figure 6. A cyclic schedule with equal bandwidth for all source-destination pairs in a 4-node network.

For clarity, we briefly describe the token flow of the schedule graph in Figure 7. Initially, the transitions AA^+, BB^+, CC^+ and DD^+ are all active and can fire in any order. As the graph is symmetric, we can discuss the flow of the actions describing only node A without loss of generality. When the event representing the beginning of a send from node A to itself occurs, AA^+ will fire, removing the two tokens on the arcs leading into the transition, and placing a token on the outgoing arc, thus activating the transition AA^-. When AA^- fires, it will place a token on each of the two outgoing arc placeholders. However, **AB^+ will not become active until BB^- has fired**. Once this occurs, both A and B have completed the first time slot, and A may proceed to the second time slot with the firing of transition AB^+. The tokens will progress down through AB^{+-}, AC^{+-} and AD^{+-}, with node A always depending on node B to finish the current time slot before it can proceed. Finally, when AD^- fires, node A will reach the initial state and be ready to repeat the sequence of firings.

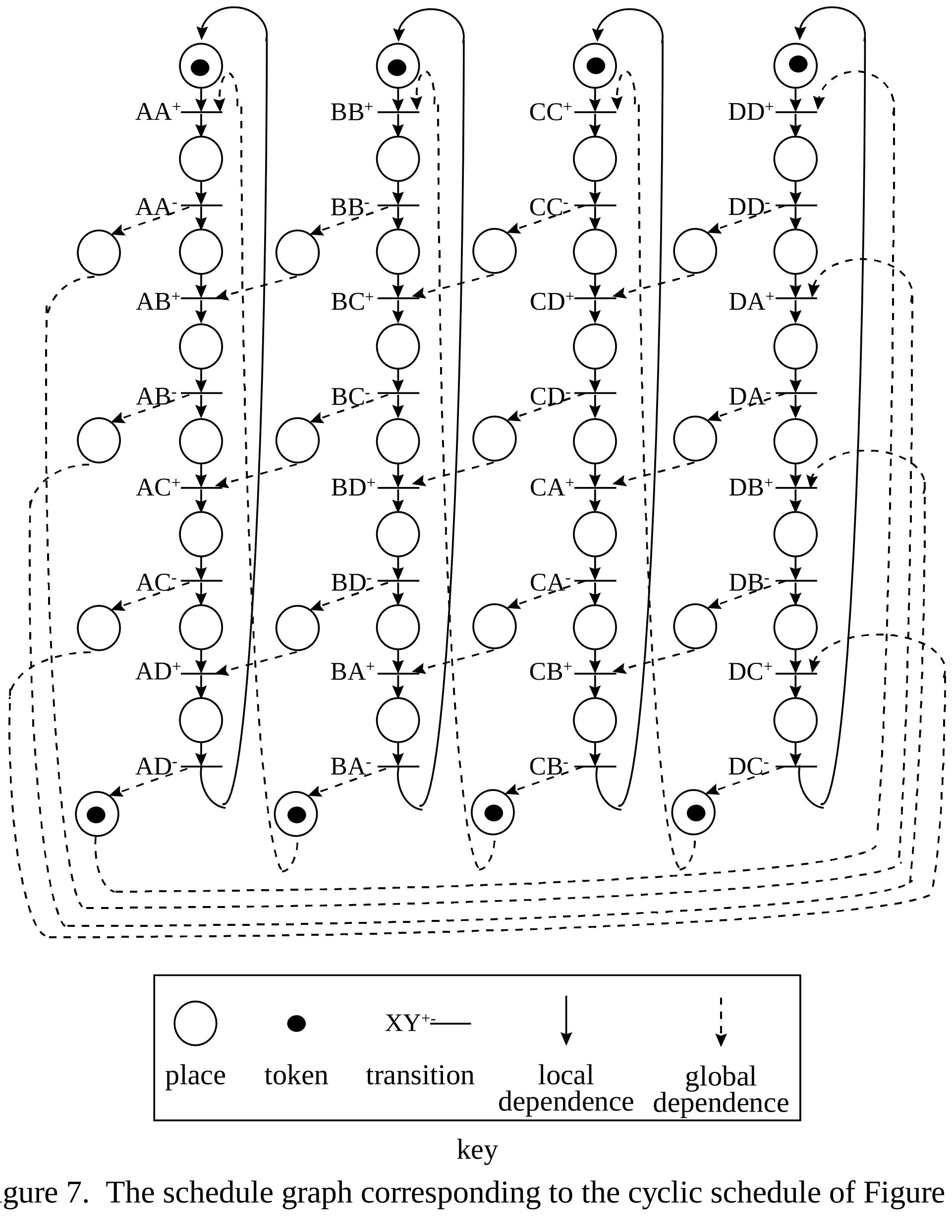


Figure 7. The schedule graph corresponding to the cyclic schedule of Figure 6.

4.2 Partitionable Schedule Graphs

The key insight in FBS is that schedule graphs can be cleanly partitioned into local dependences within a finite state machine in the sending NI and network dependences between the nodes.

A node's local state indicates when and to whom it starts and stops sending; thus, a node's local dependences consist of subsequent sends, and are easily implemented as a finite state machine on the node's NI. In a schedule graph, local dependences are represented as local arcs (solid lines) between transitions labeled XY and XZ, where X, Y and Z are nodes in the network. Figure 8 gives node A's local finite state machine for the schedule in Figure 7.

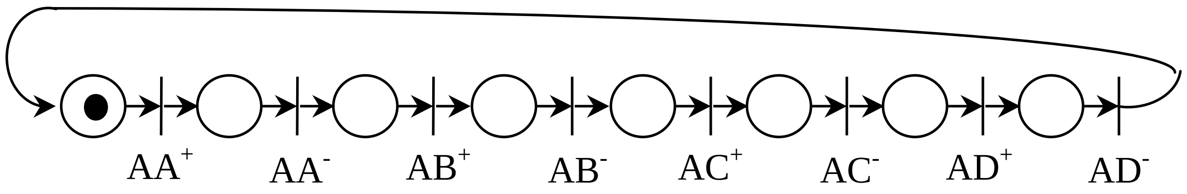


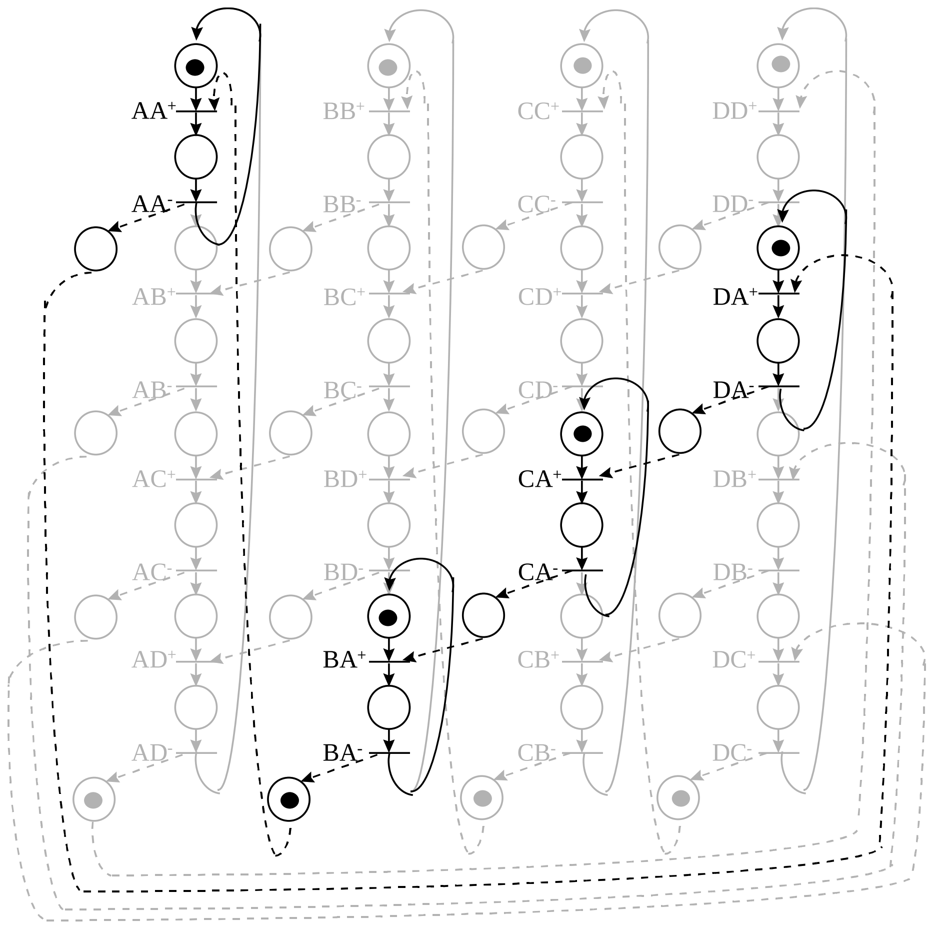
Figure 8. Node A's local finite state machine for the global schedule in Figure 7.

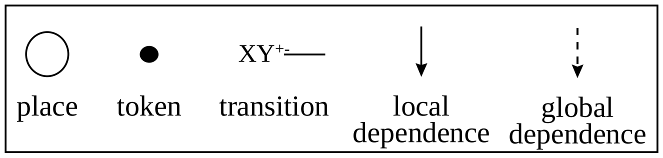
All other arcs are network arcs (dashed lines) and represent interaction between nodes. For example, in Figure 7, the arc connecting BB^- and AB^+ is a network arc. This is because node A has no local state indicating when B's message to itself has completed; this can only be determined via interaction with the network. Network arcs in a self-synchronizing graph are implemented by the local NIs detecting network feedback (hardware backpressure). This backpressure signifies that they are ahead in the schedule, and slows the local nodes' progress through the schedule. With a self-synchronizing schedule, this slowing is sufficient to keep all network nodes synchronized to the global resource schedule. Because network backpressure is not directed -- a blocked node cannot determine who is blocking it -- FBS will work only within a bounded clock skew.

4.3 Properties of Self-Synchronizing Graphs and Schedules

Consider the class of schedules (and their induced partitionable schedule graphs) that are self-synchronizing. That is, due to inherent dependences in their structure, they synchronize themselves to the slowest node in the network. In such schedule graphs, every node must indirectly depend on every other node. More precisely, for all network node pairs (X,Y), there must exist a path between XR^+ and YS^+ for some nodes, R and S. In short, their send schedules must interact through the network.

To simplify things, we restrict ourselves to a subset of self-synchronizing graphs that have at least one **synchronization cycle**, which is a cycle of dependences that includes every node in the network as a sender exactly once. Any communication schedule that contains a synchronization cycle is considered a self-synchronizing schedule. The schedule described in Figure 5 is a synchronization cycle where node D depends on node A, because D must wait for node A's message to itself in the first time slot to complete before D's messages to A in the second time slot can be received. Similarly, node A depends on B, B depends on C, and C depends on D. Thus, regardless of which of the four nodes is slowest, the dependences will synchronize all of them to the slowest node. Figure 9 gives the schedule graph for the schedule in Figure 5 as a highlighted version of Figure 7. The synchronization cycle in this figure is $AA^{+-} \rightarrow DA^{+-} \rightarrow CA^{+-} \rightarrow BA^{+-}$.





key

Figure 9: The schedule graph corresponding to the cyclic schedule of Figure 5, which is a subset of the schedule in Figure 6 (shown in gray).

4.4 Implementing Self-synchronizing Schedules

Skew will accumulate along the synchronization cycle before a dependence is encountered. It is necessary to bound this skew so that no two nodes differ in their execution of the schedule by an entire slot or more. The synchronization cycle length combined with the packet transmission time (slot size) determines the allowable variation in node clocks. Thus, given a particular variability in node clocks, we can define a synchronization schedule which has a cycle short enough that synchronization is maintained; and therefore, the desired quality of service is delivered.

More concretely, to ensure slot alignment, the maximum accumulated clock skew between nodes must be less than one slot. To be safe, we presume that each node is allowed to drift no more than ½ a slot time between synchronizations. So long as:

EQUATION 1:

(Accumulated skew) < ½ slot time

Network synchronization is maintained. However:

EQUATION 2:

(Accumulated skew) = (# slots in synchronization cycle) x (slot time) x (drift rate)

Therefore, by dividing out the slot time, the constraint becomes:

EQUATION 3:

(# slots in synchronization cycle) x (drift rate) < ½

Note that this constraint is independent of the slot size, and depends on a local (nearest neighbor) synchronization model. For large switches, log-depth methods (such as in a butterfly or FFT network) can also be used to reduce the overhead. Implications for practical systems are explored in Section 5.

5. Analysis and Performance

We first analyze the performance potential of the FBS approach, based on realistic clock qualities in current-day network interface cards. This provides a baseline for performance expectation. Then, we describe a prototype implementation which uses software emulation on the Myrinet LANai to implement the self-synchronizing schedules and report the details of its implementation and performance.

5.1 Analysis

To explore the practical feasibility of FBS, we consider the constraint of Equation 3 for the Myrinet network, ignoring at this time more sophisticated synchronization methods. In the Myrinet NI, the clock's drift rate is 300 ppm or 0.03%. Thus, the number of slots in the synchronization cycle must be less than 1/(2 x 0.0003) = 1,666. Similarly, Figure 10 reports the maximum possible number of intervening slots and corresponding synchronization overheads for a range of clock qualities.

Clock quality	# of intervening slots	% overhead for 8-port switch	% overhead for 64-port switch
200 ppm	2500	0.32	2.50
250 ppm	2000	0.40	3.10
300 ppm	1666	0.48	3.70
350 ppm	1428	0.56	4.30
400 ppm	1250	0.64	4.90
450 ppm	1111	0.71	5.45
500 ppm	1000	0.79	6.02

Figure 10: Maximum number of intervening slots and corresponding synchronization overhead for a range of network interface clock qualities.

For realistic slot times (2 kilobyte packets in our FM 2.0 implementation yield ~12 microsecond slot times), the self-synchronizing schedules can have periodicities as large as 12 to 36 milliseconds[2]. This overhead is encouraging, particularly for such a simple synchronization technique. In effect, the analysis suggests that for an 8-node Myrinet network, a self-synchronizing schedule graph can have 1,666 unregulated slots followed by one execution of a schedule similar to that shown in Figure 6, resulting in an overhead of eight slots in every 1674 slots, or 0.48%.

5.2 Myrinet Prototype

With the goal of validating the practical feasibility of FBS, we implemented a prototype for Myrinet. This system runs on a Myrinet network of 200 MHz Pentium Pro workstations with LANai 4.1 boards (see Section 2 for further details). For our initial testing, we used 2 KB packets and only four ports of the switch are active. At present, the prototype does not implement the entire FM-QoS API, but is sufficient to explore the feasibility and cost of the basic FBS approach. To this end, we use the prototype to synchronize the network nodes and then execute a self-synchronizing schedule. The prototype thus consists of two parts -- bootstrap synchronization and steady-state synchronization, where FBS is employed. The implementation of the full FM-QoS API is in progress.

5.2.1 Bootstrap Synchronization

FBS can maintain synchrony, but initial synchronization is currently achieved by a different method. Bootstrap synchronization is achieved via a standard left-right synchronization protocol amongst the NI processors (LANais) of the participating nodes. Pseudocode for the initial synchronization algorithm is shown in Figure 11. To ensure non-interference, the participating nodes must be able to communicate in a ring without re-using any network links.

```
for(i=0; i < num_nodes; i++)
{
  send_start(left);    /* Start send message to the node on left */

  while(no_incoming_message()); /* Wait until message to receive */

  recv_message();      /* Receive the message */

  while(send_not_done()); /* Wait until send to the left is done */
}
```

Figure 11: Initial NI synchronization algorithm.

Once initialized, the LANai immediately begins the bootstrap synchronization algorithm. First, it starts a send to the node on its left in the ring. Without waiting for the send to complete, it waits for and receives an incoming message from the node on its right (there is only one possible source of this message). Once it has received a message from the node on its right, it waits until the message sent to its left has been received. Then, it repeats this sequence a number of times. Because no node can be more than one iteration ahead of every other node, this algorithm achieves tight synchrony within a few iterations.

[2] For a clock skew of 500 ppm, there must be a synchronization cycle every 1000 slots. With a slot time of 12 microseconds, the cycle must occur every 1000 slots x 12 microseconds/slot = 12 milliseconds.

5.2.2 Steady State Synchronization and Validation

After bootstrap synchronization, all nodes begin executing the self-synchronizing schedule. The LANai control program implements the local finite state machine transitions as described in Section 4. This simply involves sequencing through a table representation of the schedule, sending and receiving as appropriate. Sending and receiving are overlapped in the single-threaded LANai control program, with FBS occurring as a stall on the completion of a send DMA operation.

To ensure correctness of FBS, our prototype uses extra packet header fields to verify that every packet received was sent during the corresponding slot in the schedule by the correct node. Examination of these headers indicates that the FBS approach does achieve synchrony and precisely implements the global communication schedules. Section 5.3 reports the performance of our prototype.

5.3 Performance

5.3.1 Synchronization Overhead

To measure the synchronization overhead, we used a cyclic schedule similar the one in Figure 6 and introduced empty slots between executions of the cyclic schedule. By increasing the number of empty slots until a synchronization failure occurs, we can determine the required synchronization overhead for a particular set of nodes. Note that the synchronization frequency can be chosen dynamically (even adaptively) based on the measured clock skew amongst a set of nodes.

	Slots (Self-synch Sched)	Empty Slots	Overhead
4-node	4	832	0.48%
8-node	8	832	0.95%

These results demonstrate overheads well below 1%, and clearly demonstrate the feasibility of the approach for delivering quality of service. However, the overheads are still nearly a factor of two higher than that predicted by our simple analytic model. The discrepancy can be directly accounted for by the overhead incurred by implementing FBS in software on the LANai processor rather than in hardware in a dedicated finite-state machine. For the 2 kilobyte packet size used, our prototype achieves a slot size of 23 μs. The underlying network has 1.28 Gbps links, requiring ~ 13 μs to DMA the 2 KB packet into the network[3]. The remaining 10 μs (43%) in our slot is directly attributable to software overhead. While we continue to improve our implementation, and are optimistic that this software overhead can be significantly reduced, this overhead is an artifact of the software prototype. A hardware implementation based on simple finite-state machines and table representations of schedules could reduce this overhead to nearly zero. Such an implementation would allow the use of even smaller packets with low overhead predictable communication.

[3] 2048B/packet * 1s/1. 28 Gb * 8b/B * Gb/1,000,000,000 b * 1,000,000 μs/s = 12.8 μs/packet

5.3.2 Latency Guarantees

To determine the achievable latency guarantees, we need only examine the slot sizes achieved by the prototype. Here too, the results are directly affected by the software implementation of the prototype. With a demonstrated slot time of 23 μs, our prototype can deliver a worst case latency bound of 23 μs. In comparison, for two kilobyte packets on an 8-port Myrinet switch, the worst case latency without any special mechanisms for QoS is 104 μs. Thus FBS makes an improvement of over 4:1 in this small of a switch. Software tuning and/or a hardware scheduler implementation could improve this ratio to approximately n:1, for larger n x n crossbar switches.

Further, for a non-QoS messaging layer, latency bounds for a network with cascaded switches increases geometrically with respect to the number of switches: $((p-1)^n + 1)t$, where p is the number of ports per switch, t is the time for a packet to traverse the network, and n is the number of switches the packet must traverse. However, because FBS avoids blocking in the network, latency guarantees provided by FBS will increase only by a some constant, c, times the time it takes to actually transmit a packet over the network, giving a bound of: ct. Thus, latency guarantees under FBS will provide a geometric improvement with respect to the number of cascading switches in a network.

6. Discussion and Related Work

Our initial results suggest that feedback-based synchronization (FBS) is a promising approach for implementing quality of service in high performance networks. Our analysis and prototype demonstrate the practical feasibility of the approach for achieving synchrony and delivering predictable communication performance with overheads below 1% of channel bandwidth. While the initial prototype's performance suffers from significant software overhead, we are optimistic that further tuning or hardware implementation on NI's can effectively eliminate such overhead.

ATM To date, many approaches to quality of service have focused on adding hardware to the routing switches in the network. For example, ATM uses link multiplexing and buffering to implement virtual circuits [3, 24] which are then the basis of quality of service. A wide range of scheduling algorithms have been proposed for implementation on ATM switches to provide QoS guarantees: Fair Queueing [7], Stop-and-Go [11], Hierarchical Round Robin [15], Delay Earliest-Due-Date [16], Jitter Earliest-Due-Date [27], and Virtual Clock [28]. The algorithms differ in their complexity, efficiency and fairness. The FBS approach does not require increased switch hardware capability to deliver quality of service; but instead, implements the complexity for coordination and scheduling at the endpoints. We believe that this approach allows simpler high-speed switches, and avoids the ATM controversies surrounding hardware flow control and reliable transfer. FM-QoS presumes link-level flow control and, based on that presumption, delivers reliable transfers to applications.

Tandem's ServerNet includes a scheduling algorithm called ALU Biasing [19, 13] to provide proportional bandwidth allocation for wormhole-routed networks at a modest increase in hardware complexity. However, because of the coupling induced by wormhole-routing, analysis of ServerNet networks indicate that achieving tight bounds for latency is difficult with only this mechanism, and tight QoS bounds may require external network support, such as the smart network interfaces proposed as part of FBS.

Gerla, et. al. describe three techniques for implementing QoS on a Myrinet network [10]. However to our knowledge, none of these approaches have been implemented. While the hardware context is similar, their approach differs significantly, focusing on much larger timescales (milliseconds) or requiring hardware enhancements to the Myrinet. The first approach involves separate subnets for QoS traffic. The second approach employs separate virtual channels for QoS traffic. Both approaches can give bandwidth guarantees; but neither address the blocking issue, and thus cannot give latency bounds better than a non-QoS implementation. Further, both approaches require network hardware enhancement specifically for quality of service (network interface duplication or addition of virtual channels). The third approach uses a rotating token protocol to limit the network to a single sender at a time. This approach implements quality of service without hardware modification, but fails to make use of the parallel switched interconnect. In contrast, FBS requires no change to the underlying network interconnect, yet achieves both high performance and parallel use of the network.

7. Summary

Feedback based synchronization (FBS) is a novel approach to achieving predictable network performance that utilizes network feedback along with self-synchronizing schedules to synchronize network communication. Combining FBS with global network resource schedules makes it possible to deliver high performance quality-of-service on switched networks with overheads of less than 1% of the total communication traffic. Empirical measurements of our FM-QoS prototype confirm this analysis. By avoiding all resource conflicts within the network, this technique is able to guarantee latencies between the NIs on the order of tens of microseconds.

We have built a prototype for this approach using a cluster of Pentium Pro workstations connected via a Myrinet network. Our prototype uses network interface firmware to implement FBS and successfully synchronizes a single-switch network. For 2 KB packets, our prototype can guarantee latencies as low as 23 microseconds, only 10 microseconds greater than the underlying network transfer time. This is a reduction of over 4x the latency bound provided by a non-QoS implementation (and larger reductions are expected for larger switches and multi-switch networks). In addition, worst-case latency bounds provided by FBS will show **geometric** improvement with respect to best-effort round robin scheduling in multi-switch networks. Our prototype indicates that for Myrinet, schedules can include over 800 slots between self-synchronizations, so our prototype can allocate bandwidth at granularities as small as 0.1 MB/s, subdividing a single Myrinet link into over 800 quality-of-service channels. Furthermore, there is no practical reason why the bandwidth cannot be further subdivided by increasing the periodicity of the schedule in multiples beyond 800. In summary, FBS is a promising new approach to providing predictable performance by combining high-speed, simple switches with an intelligent network interface.

7.1 Future Work

We have only begun to explore the wealth of questions that arise from this new approach to implementing quality of service in high speed switched networks. Below we outline some of the outstanding questions being pursued by our group and others working on this approach:

Packet Size We are currently using 2 KB packets, but many recent lean protocol layers use smaller packet sizes or even variable packet sizes to reduce minimum latency. We are exploring the extension of these ideas to this more general context.

FM-QoS We are implementing the FM-QoS API for Myrinet. This requires bridging the quality of service guarantees at the device level to the user API. We expect that the LANai firmware will remain much the same as in the prototype, with the addition of data structures for coordination with the host processor. The host processor code will include flow-control mechanisms and queue management similar to those in the current version of FM. The main new components for the host will be channel management and mechanisms to assign slots in a schedule.

Network Feedback We have demonstrated FBS on wormhole-routed networks. However, it appears that FBS can be used for a wide range of networks, so long as the buffering behavior and flow control information is predictable. Precisely characterizing the space of networks for which FBS is practical is an interesting question. Networks which provide link-level flow control such as Gigabit Ethernet are of particular interest.

Scaling Both larger switches and larger networks present interesting challenges. For larger switches, it becomes productive to consider more complex synchronization schedules (beyond the ring-style synchronization discussed here). For multi-switch networks, the aggregate buffering on the network switches will doubtless increase, slowing feedback to the network interface and reducing the efficiency of synchronization. Further, multi-switch networks will require a more complex reservation protocol and a global resource schedule which involves all switches. We believe these problems can be solved for cluster interconnects by evolving to smaller numbers of large arity switches (e.g. 16x16 crossbars or 16-port Gigabit Ethernet switches). With Myrinet, the buffering problem is not critical. A Myrinet switch has a buffer of 23 bytes, so an FM 2.0 packet of 256 bytes can span up to 11 switches while still blocking all the way back to the source NI. We further anticipate that for very large networks, techniques can be developed that will coordinate separate synchronous subnets, where each subnet employs FBS. One possible approach is to set up a static "channel" between two subnets, which will then have several QoS channels multiplexed into it.

Network Interface Complexity FBS requires scheduling functionality in the NIs. In our prototype, this is embodied in a LANai control program, but in general this is likely to be cast directly into hardware. We believe that the complexity of this hardware is modest, but the design of such hardware which embodies the right abstractions to describe schedules and executes them efficiently is an open question.

Acknowledgments

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, ARPA Order #E524 through the US Air Force Contract F30602-97-2-0121, NSF grants MIP-92-23732, and NASA grant NAG 1-613. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, Microsoft and Motorola is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

In addition, this material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Basu, V. Buch, W. Vogels and T. von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In **Proceedings of the 15th ACM Symposium on Operating Systems Principles**, December, 1995.

[2] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. K. Su. Myrinet: A gigabit-per-second local-area network. In **IEEE Micro**, Vol. 15, No. 1, February 1995.

[3] J. Boudec. The asynchronous transfer mode: a tutorial. **Computer Networks and ISDN Systems**, 24:279-309, 1992.

[4] Concurrent Systems Architecture Group, High Performance Virtual Machines (HPVM), Department of Computer Science, University of Illinois, http://www-csag.cs.uiuc.edu/projects/hpvm.html

[5] D. E. Culler, et. al. The generic Active Messages interface specification. http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps

[6] W. J. Dally and C. L. Seitz. Deadlock Free Message Routing in Multiprocessor Interconnection Networks. In **IEEE Transactions on Computing**, Vol. C-36, May 1987.

[7] A. Demers, S. Keshav and S. Shenker. Analysis and simulations of a fair queueing algorithm. In **Proceedings of ACM SIGCOMM**, pages 1-12, 1989.

[8] End to end Performance vIa Quality of Service, http://pertsserver.cs.uiuc.edu/epiq/

[9] D. Gall. MPEG: A video compression standard for multimedia applications. In **Communications of the ACM**, Vol. 34, No. 4, pages 46-58, April 1991.

[10] M. Gerla, B. Kannan, B. Kwan, P. Palnati, S. Walton, E. Leonardi and F. Neri. Quality of service support in high-speed, wormhole routing networks. In **International Conference on Network Protocols**, October 1996.

[11] S. J. Golestani. Congestion-free communication in high-speed packet networks. In **IEEE Transactions on Communications**, Vol. 39, No. 12, pages 1802-1812, December 1991.

[12] S. Hauck. Asynchronous design methodologies: An overview. In **Proceedings of the IEEE**, Vol.83, No. 1, pages 69-93, January 1995.

[13] R. Horst. Tnet: A reliable system are network. **IEEE Micro**, pages 37-45, February 1995.

[14] D. Jadav and A. Choudhary. Designing and implementing high-performance media-on-demand servers. In **IEEE Parallel and Distributed Technology**, pages 29-39, Summer 1995.

[15] C. Kalmanek, H. Kanakia and S. Keshav. Rate controlled servers for very high-speed networks. In **Proceedings of IEEE Global Telecommunications Conference**, 1990.

[16] D. Kandlur, K. Shin and D. Ferrari. Real-time communication in multihop networks. In **IEEE Transactions on Parallel and Distributed Systems**, Vol. 5, No 10, pages 1044-1056, October 1994.

[17] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In **International Symposium on Computer Architecture**, 1995.

[18] J. H. Kim. Bandwidth and latency guarantees in low-cost, high-performance networks. Ph.D. thesis, 1997.

[19] J. H. Kim and A. A. Chien. Rotating Combined Queueing (RCQ): Bandwidth and latency guarantees in low-cost, high-performance networks. In **International Symposium on Computer Architecture**, 1996.

[20] E. W. Knightly, D. E. Wrege, J. Liebeherr and H. Zhang. Fundamental limits and tradeoffs of providing deterministic guarantees to VBR video traffic. In **Proceedings of ACM SIGMETRICS**, 1995.

[21] M. Lauria and A. A. Chien. MPI-FM: High performance MPI on workstation clusters. In **Journal of Parallel and Distributed Computing**, February 1997.

[22] J-P. Li and M. Mutka. Priority based real-time communication for large scale wormhole networks. In **Proceedings of International Parallel Processing Symposium**, pages 433-438, May 1994.

[23] J-P. Li and M. Mutka. Real-time virtual channel flow control. In **Proceedings of IEEE 13th Annual International Phoenix Conference on Computers and Communications**, pages 97-103, April 1994.

[24]P. Newman. ATM local area networks. **IEEE Communications Magazine**, pages 86-98, March 1994.

[25] S. Pakin, V. Karamcheti and A. A. Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. **IEEE Concurrency**, 1997.

[26] P. Ramanathan, D. D. Kandlur and K. G. Shin. Hardware assisted software clock synchronization for homogeneous distributed systems. In **IEEE Transactions on Computers**, Vol. 39, pages 514-524, April 1990.

[27] D. Verma, H. Zhang and D. Ferrari. Delay jitter control for real-time communication in packet switching networks. In **Proceedings of TriComm'91**, pages 47-55, 1991.

[28] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In **Proceedings of ACM SIGCOMM**, pages 19-29, 1990.

[29] L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala. RSVP: A new resource ReSerVation Protocol. In **IEEE Network**, September 1993.

Author Biography

Kay Connelly is currently a graduate student at the University of Illinois at Urbana-Champaign, where she works as a research assistant in the Concurrent Systems Architecture Group. She received her B.S. in Computer Science from Indiana University in 1995. She currently holds a National Science Graduate Research Fellowship and is working towards her doctorate degree.

Andrew A. Chien is currently an Associate Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, where he holds a joint appointment as an Associate Professor in the Department of Electrical and Computer Engineering and as a Senior Research Scientist with the National Center for Supercomputing Applications (NCSA). Andrew received his B.S. in Electrical Engineering from the Massachusetts Institute of Technology in 1984 and his M.S. and Ph.D., in Computer Science, from the Massachusetts Institute of Technology in 1987 and 1990, respectively. He is the recipient of a 1994 National Science Foundation Young Investigator Award, the C. W. Gear Outstanding Junior Faculty Award (1995), and the Xerox Senior Faculty Award for Outstanding Research (1996).