NIFDY: A Low Overhead, High Throughput Network Interface

Timothy Callahan and Seth Copen Goldstein
{timothyc,sethg}@cs.berkeley.edu

Computer Science Division
University of California–Berkeley

Abstract

In this paper we present NIFDY, a network interface that uses admission control to reduce congestion and ensures that packets are received by a processor in the order in which they were sent, even if the underlying network delivers the packets out of order. The basic idea behind NIFDY is that each processor is allowed to have at most one outstanding packet to any other processor unless the destination processor has granted the sender the right to send multiple unacknowledged packets. Further, there is a low upper limit on the number of outstanding packets to all processors.

We present results from simulations of a variety of networks (meshes, tori, butterflies, and fat trees) and traffic patterns to verify NIFDY's efficacy. Our simulations show that NIFDY increases throughput and decreases overhead. The utility of NIFDY increases as a network's bisection bandwidth decreases. When combined with the increased payload allowed by in-order delivery NIFDY increases total bandwidth delivered for all networks. The resources needed to implement NIFDY are small and constant with respect to network size.

1 Introduction

An efficient interconnection network is essential for high-speed parallel computing. Although processor speeds and raw network performance have increased dramatically, network interfaces have not efficiently integrated these resources. Thus system performance has not kept pace with the performance of the individual components. In this paper we present a network interface that more closely matches modern network characteristics while remaining independent of the network design.

Interconnection networks deliver maximum performance when the offered load is limited to a fraction of the maximum bandwidth. We call this the *operating range* of the network. Many people have observed that when the offered load exceeds the operating range, throughput falls off dramatically [Jac88, Jai90, SS89, RJ90, KS91, Aga91, BK94]. Researchers have investigated this problem for both WAN and MPP networks. The WAN solutions are based on deep networks with long messages and generally use software protocols at the end points [Jac88, RJ90, KMCL93, SBB+91]. Most MPP networks, which are shallower and have shorter messages, either ignore the issue or control congestion in the network fabric itself [CBLK94, Dal91, LAD+92, Dal90]. In this paper we propose a network interface called NIFDY—**N**etwork **I**nterface with **F**low-control and in-order **D**eliver**Y**—which adapts the WAN style solutions to MPP networks. In short, NIFDY performs *admission control* at the edges of the network; a packet is injected into the network only if the destination is expected to be able to accept the packet.

When the network is running within its operating range, software overhead represents the largest cost in message transmission. Some of this overhead arises in matching the functionality of the network fabric to the application requirements. NIFDY removes the overhead required for reordering packets by delivering packets to the processor in the order in which they were sent. This allows network designers to exploit various techniques, e.g. adaptive routing, to increase network performance without imposing additional overhead on the applications.

In the rest of this section we explain our assumptions about the underlying network and present the basic design of NIFDY. In Section 2 we present the complete design of NIFDY, its implementation cost, and how it interacts with the processor and network. Section 3 describes the simulator which was used to verify the performance of NIFDY and Section 4 presents the results we obtained from it. In Section 5 we compare our approach to previous work on network design. In Section 6 we propose some extensions to NIFDY to handle unreliable networks and networks of workstations.

1.1 The Underlying Network

NIFDY is designed primarily for MPPs where the processors are tightly coupled by a fast, shallow interconnect. Unlike previous work to increase performance of such systems, our approach does not presuppose a particular kind of network or router. We assume only that once the network has accepted a packet it will eventually be delivered to its destination, if processors continue to accept packets. (In Section 6 we show how NIFDY can be extended to handle unreliable networks.)

On most networks proposed for MPPs, the main source of performance degradation is congestion. Congestion can be caused in two places: at the end-points and internally in the network fabric. End-point congestion arises when packets arrive at a node faster than the node can process them. Internal congestion can arise for several reasons. First, a number of senders may combine to generate traffic that exceeds the network's bisection bandwidth. Second, hot spots in the network may cause unnecessary blocking and reduce utilization. Third, faults in the network may restrict

Appears in *Proc. 22nd Annual International Symposium on Computer Architecture, June 1995, Santa Margherita Ligure, Italy.*

the available bandwidth. Finally, end-point congestion can cause congestion internal to the network; we call this *secondary blocking*.

To handle hardware faults and transient congestion, many network topologies—e.g. fat trees and multibutterflies—provide multiple paths that spread out traffic between nodes. While networks with alternative paths can provide some congestion tolerance, they don't solve the problem entirely and can even aggravate it. If there is no direct feedback to the sending node (as is the case in most, if not all, MPP networks), then backpressure is the only mechanism to stop the sender from sending packets. In this case, adaptive routing may fill up the network buffers along all possible paths between the sender and the bottleneck, causing extreme secondary blocking.

To avoid secondary blocking, when a node is sending to a blocked or overloaded receiver it must stop or slow its injection of packets into the network. Two schemes have been proposed to accomplish this: rate-based flow control (RBFC) and credit-based flow control (CBFC). RBFC limits each sender to a rate that is known not to induce secondary blocking, assuming the receiver is pulling packets out of the network. CBFC gives each sender a credit of packets that it can inject before secondary blocking will happen. The problem with both these schemes is that MPP traffic is bimodal—processors are usually sending either at full speed or hardly at all. With RBFC a fixed rate will not properly utilize the network; for instance, the optimal rate when only one sender is active is different from the optimal rate when all senders are active. While CBFC solves this problem, it does not eliminate secondary blocking if many senders have accumulated credits and simultaneously send a burst of packets. In addition, RBFC requires negotiation between the sender and the receiver or requires global information to be maintained in the network, while CBFC requires overhead for maintaining the credits, possibly on a per-receiver basis. These costs, combined with the bimodality of MPP traffic, have prevented designers from using RBFC or CBFC in MPP networks.

The price of randomized routing techniques is that packets may be delivered out of order. Even meshes and tori using dimension-order routing may deliver packets out of order if they utilize multiple virtual channels to alleviate congestion [Dal90]. For medium-sized transfers on the CM-5, [KC94] showed that reconstructing the original transmission order accounted for as much as 30% of the total transfer time. The Synoptics ATM Switch routes packets adaptively within the switch and then reorders them before they leave the switch. This link-by-link reordering increases the latency of the switch by a factor of five [BT89]. From these observations we conclude that reordering should be performed only once, at the destination, and that if possible the reordering should be performed in hardware.

1.2 NIFDY in a Nutshell

NIFDY is a network interface that uses admission control to perform both in-order delivery and end-to-end flow control. To handle the bimodality of MPP traffic and ensure in-order delivery, NIFDY has two communication modes. The default case is *scalar mode* in which only a single packet can be outstanding to a given destination processor. To allow for higher bandwidth communication, a processor can request *bulk mode* which, if granted, gives the sender extra credits that can be used for communicating only with the granting processor.

For every scalar packet sent, the destination processor number is recorded in an *outstanding packet table* (OPT). Until an acknowledgment (ack) is received from the destination and the entry in the the OPT is cleared, NIFDY will not inject any further packets destined for that processor. However, if the OPT is not full, it can send a packet to another processor. This keeps packets from one processor to another in order. Furthermore, it reduces end-point congestion and adjusts to hot-spots, the bisection bandwidth, and possible faults. For shallow networks, the round-trip latency is smaller than the time it takes to inject a packet into the network and no extra latency is noticed even for consecutive sends to the same destination.

By keeping the OPT small enough, we can adjust for network volume, ensuring that secondary blocking is reduced. If a processor is not responding to the network, each processor will send at most one packet to it; no further packets will be sent until the destination processor wakes up and accepts a packet. At this point NIFDY will send an ack, allowing another packet to be sent. NIFDY also incorporates an outgoing buffer pool which reduces head-of-line blocking in the network interface. Thus, if several messages are ready to go to different processors, they can be interleaved up to the limit of the OPT.

To accommodate deeper networks and large round-trip times, the NIFDY protocol has a transfer mode in which multiple unacknowledged packets can be in transit between two processors. In the case where the sender has multiple packets to be sent to a single destination it can request a *bulk dialog*. If the receiver grants such a dialog in the ack, then the sender can send more than one packet per ack. By limiting the number and size of bulk dialogs a receiver will grant, we can again limit secondary blocking even for bimodal traffic.

In short, NIFDY implements a simple extension to network interfaces that allows increased flexibility in network design while limiting congestion and decreasing software overhead. NIFDY's resource requirements increase with desired performance, not with the number of nodes in the machine.

2 The NIFDY Unit

NIFDY is a network interface that increases system performance by decoupling the processor and the underlying network fabric. The processor sends packets by inserting them into NIFDY, then NIFDY takes over and injects them into the network at the earliest opportunity, according to the protocol described below. NIFDY handles flow control, ordering of packets, and, if extended for unreliable networks, packet retransmission. In this section we describe the basic design, the parameters to tune NIFDY to match the processor and the network, and the implementation costs of NIFDY. The ideas in NIFDY can be added to any network interface.

NIFDY distinguishes two types of network data packets, scalar and bulk. Scalar packets are best used for short messages while bulk packets are best used for large block transfers. In addition, NIFDY generates acknowledgment (ack) packets, which are used to keep packets in order and to provide access control. Every scalar packet is acked individually and bulk packets are acked using a sliding window protocol. The ack packets share the same network as the data packets, but are consumed by the receiving NIFDY.

Each processor can send only one scalar packet at a time to any other processor. For every scalar packet sent, the destination processor number is recorded in an *outstanding packet table* (OPT). Until an acknowledgment is received from the destination processor and the entry in the the OPT is cleared, NIFDY will not inject any more packets bound for that processor. However, if the OPT is not full, it can send a scalar packet to a different destination.

Clearly, there is no way that packets can become misordered in the network if there is at most one outstanding packet between each sender/receiver pair at any instant. The basic flow control is also evident: If the receiving node is ignoring the network, or for some

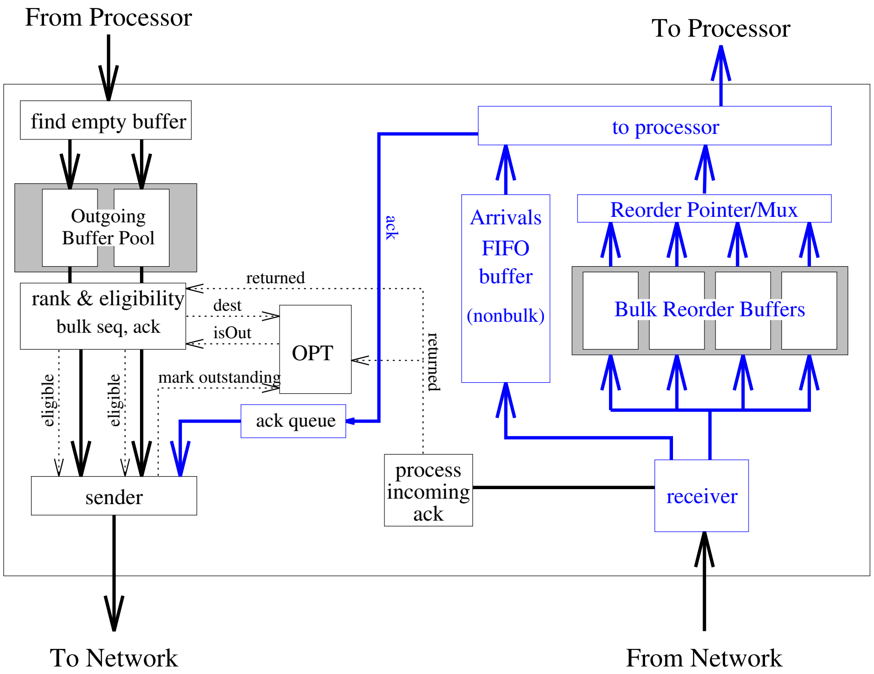

Figure 1: *Block diagram of the* NIFDY *unit with support for bulk dialogs.*

other reason is not pulling packets out of the network rapidly, the sender will not get its ack and will refrain from sending any more to that node. This also provides a mechanism for congestion control within the network; if the packets or acknowledgements between a sender/receiver pair must cross a hot spot, the round-trip delay (and thus the delay between consecutive packets sent to the same destination) will increase, throttling the bandwidth of conversations and reducing congestion.

The restriction of having only one outstanding packet may seem excessive at first, but for the types of low-latency tightly-coupled multiprocessor networks we are considering, it has little effect on throughput.[1] We are relying on the fact that wormhole or cut-through routing will be used, so that in the absence of contention, the head of the packet can often reach the destination before the tail has even left the source. Since the ack can be sent as soon as the header of the incoming packet is processed, in many cases the sender will receive an ack for the packet it is currently sending. We will look at this issue more in Section 2.4.

When a network has a high round-trip latency, sending multi-packet messages as scalar packets may not fully utilize the network. We overcome this by sending multi-packet messages using the bulk protocol. In this protocol, the sender requests a *bulk dialog*, which, if granted, allows the sender to have more than one packet outstanding to the destination. Although the network may deliver the multiple outstanding packets out of order, the receiving NIFDY puts them back in order before presenting them to the processor.

Instead of piling up in the network, packets are blocked in the sender's NIFDY. This reduces secondary blocking and increases throughput. Since both the packet and the ack have to traverse the network, any hot spot or network congestion will slow down both, delaying injection of more packets into the network. NIFDY usually reacts to a slow receiver or network congestion long before packets back up all the way to the node's network port; this has the key benefit that NIFDY can start sending to other ready destinations. By contrast, if backpressure is the only way of telling when to slow down, a sender will continue injecting packets to a slow receiver until its entrance to the network is blocked, at which point it is usually blocked from sending to any other destination. In fact, we expect that by reducing secondary blocking NIFDY will enhance the value of adaptive routing, since alternative paths will be available more often.

2.1 Protocol Implementation

Networks have different characteristics which affect the amount of traffic that they can handle before congestion reduces throughput. Thus, for best performance, NIFDY will have to be tuned for each network. This is done by adjusting four parameters.

O**:** Size of outstanding packet table (OPT).

B**:** Size of the outgoing buffer pool.

D**:** Maximum number of bulk dialogs each receiver can maintain simultaneously.

W**:** Receiver window size for the bulk dialog protocol.

For most shallow networks, the most important parameters are O and B. If the OPT is large, then the processor can have more outstanding packets in the network. To reduce head-of-line blocking at the sending NIFDY unit, there can be a pool of buffers to hold outgoing packets. As long as the OPT is not full, any eligible packet in the pool (we define eligibility below) can be sent. This allows the processor to interleave small packet streams for multiple processors.

The parameters D and W determine the number and size of bulk dialogs. Each sender can maintain only one outgoing bulk dialog, although it can send packets in non-bulk mode to other destinations concurrently with a bulk dialog. Each receiver can maintain D

[1] In fact, the number of outstanding messages per processor for these networks under lightly loaded conditions is often less than one [Cul94].

incoming bulk dialogs, each with a different sender. For each bulk dialog, W packet buffers are available in hardware at the receiver to provide storage for the sliding window protocol.

2.1.1 Scalar Packets

Figure 1 is a block diagram of the NIFDY unit. (This figure also shows extensions for the bulk protocol, which will be explained later.) Packets enter NIFDY from the processor if there is an empty buffer in the outgoing pool. To maintain the correct transmission order of packets to the same destination, the *rank/eligibility unit* ranks each packet in the pool relative to the other packets for the same destination. The rank value indicates how many other packets there are in front of it. When a packet arrives at the pool, its rank is assigned based on the contents of the pool and the OPT: the rank is one plus the number of waiting and outstanding packets for the same destination. Whenever an ack from a processor is received, all packets in the pool to the same processor have their rank decremented by one, bringing to zero the rank of the next packet to be transmitted (making it "eligible").

When the network can accept another packet, and there is a free entry in the OPT, and there is at least one eligible packet in the buffer pool, then one of the eligible packets is chosen for sending. The chosen packet is injected into the network, and the destination processor number is recorded in the OPT. Until an ack is received from that processor, no further packets are eligible for transmission to it. Note that every packet must contain the source processor ID in its header so that the destination processor can return an ack.

When a data packet is received from the network, it is inserted into the arrivals FIFO buffer. When it is accepted by the processor an ack is returned.[2]

2.1.2 The Bulk Protocol

Figure 1 also shows how NIFDY handles bulk dialogs. The header of each packet includes a bulk-request bit. A sender requests bulk mode by setting the bulk-request bit in the header of a non-bulk packet. The receiver grants bulk mode to the sender by including a bulk dialog number in the ack it returns. A receiver may maintain multiple bulk dialogs, so it must give each active sender a different dialog number. If the receiver can't grant bulk mode because it is already participating in the maximum number of bulk dialogs, the ack will indicate the rejection to the requesting sender. In this case, the sender will continue sending its data using scalar packets, and can continue requesting bulk mode, which may eventually be granted if a bulk dialog slot becomes available.

When a node sends to a receiver that has granted it a bulk dialog, it does not insert the receiver's ID into the OPT; instead the rank/eligibility unit tracks the outstanding packets for the bulk dialog. The multiple outstanding packets may arrive at the receiving NIFDY out of order; hardware buffers provide a place to store such packets until the intervening ones arrive. Packets that arrive in order are not held up and can be streamed to the processor immediately via cut-through buffering. Sequence numbers, which need only be as large as W, are included in the header of each packet to provide ordering information. A $\{$*sequence number, dialog number*$\}$ pair replaces the bits that would have been used as the source identifier. The NIFDY unit at the receiving end replaces the dialog number in the header with the source identifier before giving the packet to the processor.

[2] An alternative, but surprisingly less effective, strategy is to send the ack earlier, when the packet is inserted into the arrivals FIFO.

A sender exits bulk mode by setting a *bulk exit* bit in the header of the last packet. A receiver can also terminate a bulk dialog in which case the transmission continues in scalar mode.

2.2 Software Issues

To get full performance out of NIFDY, the software communication layer must take into account three features of NIFDY. First, the processor must initiate bulk mode requests; NIFDY won't attempt bulk mode on its own.[3] Second, every packet includes its source node address. Finally, packets are delivered in the order in which they are sent.

In order to utilize the bulk mode of NIFDY, the communication layer will have to turn on the bulk-mode request bit in the header of outgoing packets. The designer will have to decide what size transfers will request bulk mode. If the size is too small, the resources might go to the wrong sender. If too large, unnecessary delays will result.

Since the NIFDY protocol requires an ack to be returned to the sender, the sender's address is encoded in the header of every packet. If this is exposed to the receive handlers, then the source node never needs to be included in the data portion of the packets. For instance, 51% of the request messages in the Split-C library include the source processor ID in the message. The generic active message specification requires that all request messages include the source ID [Mar]. In all these cases, the source ID required in the packet header by NIFDY could be put to good use. Thus, NIFDY's requirement of including the source ID in every packet does not actually increase overhead.

Because the messages are delivered in order, large transfers can be accomplished without requiring a round trip to initialize the destination processor's data structures or buffers. The first message can initialize the destination processor while subsequent messages contain the data. The payload per packet is increased because later packets need not include any bookkeeping information.

If the extension in Section 6.1 is implemented, then messages that expect replies could be marked as not requiring an ack. Instead, the reply itself would serve as an ack. The reply could also be marked as not needing an ack, reducing the overhead of acks to those cases where the sender is unsure whether the receiver can respond.

2.3 Implementation Cost

Aside from the control logic, which is relatively small in terms of chip area, there are three sets of buffers and two content-addressable memories needed to implement NIFDY. The buffers can be implemented using single-ported RAM, taking up less area per bit than typical three-ported register files. Thus, the $DW + B$ buffers needed can be implemented in a small space.

In order to implement the outstanding packet table, a small content-addressable memory is required. The memory contains only the tags, which must be long enough to contain the node identifiers. The number of tags is equal to O, the maximum number of outstanding scalar packets. As shown in Section 4.2, eight is usually more than sufficient. If we assume that 16 bits are enough for node identification (allowing 65536 different nodes), then we have a 16-bit by 8-entry content-addressable memory. The rank determination logic also requires a small CAM of size log the number of nodes (e.g. 16) bits by B (e.g. 8) entries.

[3] Of course, NIFDY could be extended to set the bulk-mode request bit automatically based on the locally observed traffic pattern; we have not investigated this possiblity in depth.

Parameter	Meaning
P	Number of nodes
d	Distance to destination in hops
w	Packet payload in bytes
T_{send}	Total time for processor to send packet (software overhead)
$T_{receive}$	Total time for processor to receive packet (software overhead)
T_{link}	Total time for one packet to cross a link along the path from source to destination in the absence of contention (i.e. hardware bandwidth limitation on interpacket arrival times)
$T_{ackproc}$	Total latency involved in generating and processing ack
$T_{roundtrip}$	Total latency from the time the header of the packet leaves NIFDY unit to the time the ack has been processed

Table 1: *Network characteristics influencing selection of* NIFDY *parameters.*

2.4 Parameter Selection and Performance Analysis

Initial estimates of the parameters for the NIFDY unit can be obtained by considering some parameters of the connected network and the expected traffic distributions. We will consider many distributions in Section 4. Here we give a flavor of how NIFDY would be tuned to a network by looking at network parameters and traffic between a single source/destination pair separated by d hops. Table 1 defines the parameters we are using. Without the NIFDY unit, the maximum bandwidth between two nodes in the network is

$$Bandwidth = \frac{w}{\max(T_{send}, T_{receive}, T_{link})} \tag{1}$$

which expresses that the bandwidth can be limited by the send overhead, the receive overhead, or the physical bandwidth.

2.4.1 Scalar Mode Parameters

When the NIFDY unit is included, the critical network parameter is packet latency. In most networks this latency is a function of d, the number of hops between the nodes, so we will write latency as $T_{lat}(d)$. The time from when a packet starts leaving until the ack is received and processed, defined as $T_{roundtrip}(d)$, can be calculated as

$$T_{roundtrip}(d) = 2T_{lat}(d) + T_{ackproc} \tag{2}$$

where $T_{ackproc}$ is the time it takes the NIFDY unit to generate and process the ack at both ends. Because the sending node must wait until it gets the ack before sending the next packet to the same node, packets can be sent no faster than once every $T_{roundtrip}(d)$ cycles. To attain full bandwidth between two nodes separated by d hops using the basic NIFDY protocol (with no bulk dialogs), we need $T_{roundtrip}(d) \leq \max(T_{send}, T_{receive}, T_{link})$.

2.4.2 Parameters for Bulk Dialogs

When bulk dialogs are included because pairwise bandwidth would be unnecessarily limited using the basic protocol, we can use similar calculations to decide the size of the window. For simplicity, we will assume that $T_{receive}$ is the limiting factor. If this is not the case, then T_{link} or T_{send} would be substituted.

We use a sliding window protocol in which acks are combined so that only one ack is sent for every $W/2$ packets. (Recall that W is the receiver window size.) In this case an ack will be sent only when all of the packets in that half of the window have arrived; to avoid bandwidth restriction, this ack must get back to the sender before all of the packets in the other half of the window have been injected. The round-trip time must be less than or equal to the injection time for $W/2+1$ packets. (The $+1$ is there because the round-trip time overlaps with the injection of the last packet from the other half of the window.)

$$\begin{aligned} (W/2+1)T_{receive} &\geq T_{roundtrip}(d) \\ W &\geq 2(T_{roundtrip}(d)/T_{receive} - 1) \end{aligned} \tag{3}$$

We could instead have used a sliding window protocol in which every packet is acknowledged as it is received. In this case, to reach maximum throughput we have

$$W \geq T_{roundtrip}(d)/T_{receive} \tag{4}$$

D, the parameter controlling the number of bulk dialogs per receiver, is normally set to one. However, in the unlikely event that the send rate is much slower than the receive rate, it would be desirable to increase D to the maximum point at which one receiver can handle D senders without falling behind.

When choosing parameters for the bulk dialogs, performance under light traffic loads must be balanced with performance under heavy traffic loads. Less restrictive parameters (more bulk dialogs, larger windows) will give better performance with light traffic but may lead to excessive congestion when all processors try to send simultaneously. More restrictive parameters will give better, more predictable performance with heavy traffic, but may unduly restrict light traffic.

Network characteristics determine at what point generous NIFDY parameters lead to congestion. A small network volume means that a few extra packets will cause congestion more quickly. Also, a small bisection bandwidth means that excess packets are more likely to get blocked within the network, compounding congestion. Note that if a slow receiver, rather than bisection bandwidth, is the bottleneck, bulk packets will wait in the reorder buffers and not add to network congestion.

2.4.3 Example Network Parameters

In this section we try to estimate good NIFDY parameters for two specific networks. We will assume that the NIFDY processing takes 2 cycles at each end, for a total of $T_{ackproc} = 4$. We will also assume that the T_{send} is 40 cycles and $T_{receive}$ is 60 cycles.

First we look at an 8-by-8 mesh using wormhole routing. Multiple virtual channels are not needed because it is a mesh, not a torus. The flit size used is one word (32 bits), and each flit buffer holds at most two flits. Our simulated mesh had a one-way latency of $T_{lat}(d) = 4d + 14$. With uniform traffic, the maximum and average internode distances are 14 and 6 hops respectively; hence Equation 2 gives maximum and average roundtrip latencies of 144 and 80 cycles respectively.

Since the limiting factor without the NIFDY unit would be the 60-cycle receive overhead, it is clear that the roundtrip latency of the basic NIFDY protocol will often be the limiting factor in pairwise bandwidth with an uncongested network. Thus it appears that using a bulk dialog may help. Equation 3 indicates that in order to hide the maximum NIFDY roundtrip latency of 144 cycles, we will need a bulk window size of $W \geq 2(T_{roundtrip}(d)/T_{receive} - 1)$. So we would want at least 2 packets, possibly 3 or 4 if we can afford to be generous.

This wormhole mesh has an exceptionally low volume—eight words per node (two words for each incoming link). Thus even if each node has only one eight-word packet in the network, the network will be full. This, combined with the mesh's low bisection bandwidth of $\sqrt{P}$, leads to a conservative decision regarding how many packets to allow on the network. An initial guess would have $O = 4$, $B = 4$, $D = 1$, and $W = 2$.

The other network we will consider is a full 4-ary fat tree of 64 nodes. With three levels of routers, the maximum internode distance is 6 hops, and the average distance is not much less than that. In this case $T_{lat} = 5d + 2$, giving a round-trip latency of $32 + 32 + 4 = 68$ cycles. Thus it appears that the basic NIFDY protocol may be sufficient, and bulk dialogs will help only marginally.

Our simulated fat tree's volume is 10 buffers per node, much greater than that of the mesh. This large volume, along with the fat-tree's large bisection bandwidth, means we can be less restrictive in allowing packets into the network. Thus, although bulk dialogs are only marginally useful, they probably won't hurt much either. The main effort should be to reduce the restrictions on scalar packets as much as possible. This can be done by making the OPT large ($O = 8$ entries) and by making the buffer pool for waiting packets large ($B = 8$ buffers) to reduce head-of-line blocking.

3 Simulation

Empirical results were gathered using a parallel simulator written in C++ and executed on a Thinking Machines CM-5. In these experiments, the simulated objects are distributed across the CM-5 nodes and connected using links provided by the simulator framework. Most simulation parameters are supplied at run time, allowing easy exploration of the design space.

Each cycle is simulated explicitly and synchronously by all objects; at any time in the simulation, all objects have executed up to the same point. The only exception is when real Split-C programs are driving the simulator. In this case, the network simulations on the CM-5 nodes are still synchronous, but the computation on each node is allowed to run ahead (in simulated time) up to the next point where it interacts with the network. Synchronization between the network simulation and the computation is simplified because only polling message reception is allowed; thus the computation always initiates interaction with the network.

The simulator supports the following networks (as well as others not used in this paper[4]).

- Two- and three-dimensional meshes and tori utilizing wormhole routing with virtual channels. The size in each dimension, the number of virtual channels, and buffer sizes are all run-time parameters. Links were one byte wide for all simulations reported here.

- 4-ary fat tree with 1-byte links, using either cut-through or store-and-forward routing.

- Fat tree more similar to the CM-5 [LAD+92]. Routers in the first two levels are connected to two parents rather than four, reducing bisection bandwidth as compared to a full 4-ary fat-tree. Also, the link bandwidth was reduced to 4 bits per cycle as in the CM-5 network.

- Multibutterflies, with adjustable dilation and radix. In this report we use a butterfly (dilation 1, radix 4) and a multibutterfly (dilation 2, radix 4).

All topologies support two logically independent networks, the *request* network and the *reply* network, in order to deal with fetch deadlock. With all topologies other than the CM-5 fat tree, the two networks are demand-multiplexed over the same physical links in order to make use of all available bandwidth even when the traffic is unevenly divided between the two logical networks. With the CM-5 fat tree, the two networks are strictly time-multiplexed every other cycle, so that each network is limited to eight bits every two cycles regardless of the traffic on the other network.

The simulator supports the following traffic loads.

- Pseudo-random, bursty traffic. Burst length distributions are adjustable, global barriers can be included between send bursts, and nodes can programmed to enter 'non-responsive' periods during which they neither send packets nor pull them from the network interface. Dedicated state for each pseudo-random number generator ensures that the same sequence of bursts is generated regardless of network and NIFDY configuration used. Packet size is eight words including header.

- The cyclic-shift all-to-all communication pattern described in [BK94]. This and the following two traffic patterns use the CMAM and Split-C libraries from the CM-5, and thus use six-word packets (for *all* networks, not just the CM-5 imitation).

- EM3D, an irregular electromagnetics application [CDG+93].

- Radix sort, which uses single-packet messages for both counting the keys and transferring each key to its appropriate destination [Dus94].

When the NIFDY units are included in the simulation, all NIFDY parameters are adjustable. An option allows the NIFDY units to be included but disabled. In this configuration, the extra buffering in the outgoing message pool and the arrivals queue of the NIFDY units can still be utilized. This allows us to separate the effects of the NIFDY protocol itself from the benefit of simply having extra buffering. When comparing NIFDY to buffering only, the same total amount of buffering is always used, although in order to make the fairest comparison it is redistributed to be most effective for each case. For example, with the NIFDY protocol, the capacity of the arrivals queue is at most two packets; without the protocol, best performance results from allocating at least half of the total buffering resources to the arrivals queue. Of course, the acks used in the NIFDY protocol are included directly in the simulations, competing with data packets for network bandwidth.

For realistic timings on our simulations, we ran several tests on a real CM-5 to estimate packet sending and receiving overheads as well as CM-5 network latency and bandwidth. These parameters, summarized in Table 2, agree closely with those reported in [vE93].

4 Results

4.1 Synthetic Workload

To learn which NIFDY parameters were best for which networks and to measure the overall effectiveness of NIFDY's flow control, we ran many simulations for each network. Because performance at both heavy and light network loads is important, we used two different traffic patterns for these runs: one which rewards graceful handling of heavy traffic loads, and one which rewards rapid packet delivery under light traffic.

[4]The simulator is available at ftp://ftp.cs.berkeley.edu/pub/packages/nifdy/nifdy.html

Operations	Processor cycles
Active message send	46
Active message poll (no message)	22
Active message receive (dispatch, handle, return)	56
One-way latency (incl. software) from send to beginning of handler	140 + 8 * h

Table 2: *Measured CM-5 parameters used in our simulator.*

Both traffic patterns consist of phases separated by barriers. A node that is sending during a phase will attempt to send its packets (typically 100 to 300 of them) as quickly as possible. Processors send single- or multi-packet messages; all the packets in a single message are sent consecutively and to the same destination. At the end of a message, a sender randomly chooses a new destination and message length and immediately starts sending to the new destination.

To ensure that every node makes progress sending, no node can start the next phase until all sending nodes complete the current phase. As with real MPP bulk-synchronous applications, if some nodes are favored by the topology and are able to send their outgoing data quickly, sooner or later they will have to wait until the other nodes catch up. In a bulk-synchronous application the bottom line is how quickly each communication phase is completed; thus our metric is the number of packets delivered within a fixed number of cycles. Note that this metric measures only the benefit of reduced network congestion; the in-order delivery provided by NIFDY will give an additional bandwidth benefit, dependent upon the application (see Section 2.2).

In the heavy traffic pattern, all nodes send each phase, and message lengths — the number of consecutive packets a processor sends to its destination before changing to a new destination — are uniformly chosen from one to five packets. In the light traffic pattern, each node has only a 33% chance of sending each phase, reducing contention in the network. Since nodes are less likely to poll during light traffic, our simulated nodes periodically ignore the network; these periods of ignoring the network are triggered pseudo-randomly and independently for each node. With light traffic the message length distribution includes lengths of 10 and 20 packets; most messages are short, but long messages account for more packets overall. Thus the light traffic benchmark mainly measures pairwise bandwidth with only some contention in the network and some possibility of target collisions (multiple nodes sending to the same receiver) and unresponsive receivers.

Figures 2 and 3 show the performance benefit of NIFDY for various networks under both traffic loads, comparing no NIFDY; buffering only (without the NIFDY protocol); and NIFDY using the best set of parameters for that network. The graph compares packet throughput for each case, showing the benefit just from the reduced network congestion allowing more packets to get to their destinations. For the networks that deliver packets out of order, the actual benefit of NIFDY will likely be greater; NIFDY's in-order delivery can result in more payload per packet in multi-packet messages, and can also reduce the receive processing time.

The best NIFDY parameters, chosen to give the best average performance with both test traffic patterns, are shown in Table 3. The ideal NIFDY parameters for the fat-tree variations are less restrictive than those for the meshes; fat trees have greater bisection bandwidth, greater volume, and more alternative paths between nodes, so that having a few extra packets in the network does not hurt as

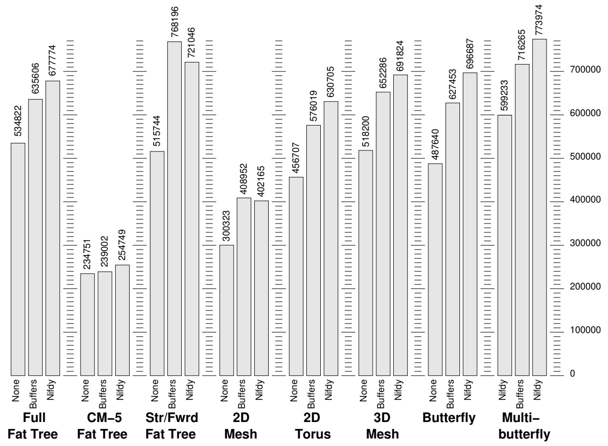

Figure 2: *Performance benefit from flow control of* NIFDY *for different networks: packets delivered in 1,000,000 cycles. "Heavy" synthetic traffic. Does not reflect additional benefit of in-order delivery from* NIFDY.

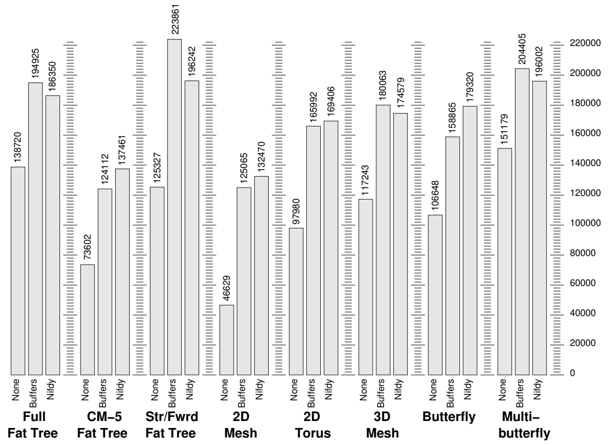

Figure 3: *Performance benefit from flow control of* NIFDY *for different networks: packets delivered in 1,000,000 cycles. "Light" synthetic traffic. Does not reflect additional benefit of in-order delivery from* NIFDY.

much as it does with the mesh. The CM-5 network has smaller bulk windows than the full fat tree even though the round-trip latency is twice as great; this is because of the CM-5 network's smaller volume and bisection bandwidth, which makes congestion a more important factor. Finally, observe that the butterfly is the only network where it is best to have no bulk dialogs: every packet travels only three hops, resulting in very low round-trip latency, and there are no alternative paths between nodes, making congestion avoidance more critical.

4.2 Scalability

Is it necessary to increase the size of the OPT or the outgoing buffer pool (O or B) as the number of nodes in the network gets larger in order to maintain the same relative benefit from NIFDY? This would be an undesirable finding, since we would like NIFDY to be scalable—we don't want to have to make all the NIFDY units bigger

Network	$T_{lat}(d)$	Maximum Volume (packets per processor)	NIFDY Parameters			
			D	W	B	O
Full Fat Tree	$5d+2$	10	1	4	8	8
CM-5 Network	$9d+2$	6.5	1	4	8	8
Store-&-Forward Fat Tree	$(L+1)d+3$	10	1	8	16	8
8x8 Mesh, 1 virt. ch., 2-flit buffers 8x8 Torus, 2 virt. ch., 4-flit buffers	$4d+14$	1 4	1	2	8	4
4x4x4 Mesh, 2 virt. ch., 4-flit buffers	$4d+18$	6	1	2	8	4
4-ary Butterfly	$5d+2$	6	0	0	8	8
4-ary Multibutterfly	$5d+2$	10	1	4	8	8

Table 3: *Characteristics of simulated 64-node networks along with best* NIFDY *parameters for each network, used in Figure 2 and 3. d is the number of hops, L is packet length in bytes.*

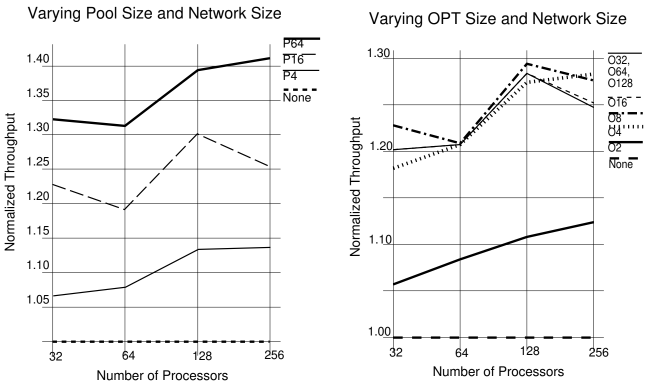

Figure 4: *Throughput for various O and B on various sized fat-trees.*

when we increase the number of nodes in our MPP.

To answer this question we ran some simulations of the full fat tree, using only short messages and no bulk dialogs in order to concentrate on the effects of O and B. The first part of Figure 4 shows throughput (normalized to a network without NIFDY) vs. machine size for different values of B. In general, increasing B gives better performance for any size network. However, for a fixed B, the relative benefit of NIFDY does not decrease and in most cases increases as the size of the MPP grows. This result means that a system designer can choose B once, depending on the desired performance and cost, and then can expect to maintain that performance benefit even as the MPP scales to large sizes.

The second part of Figure 4 shows normalized throughput vs. machine size for different values of O. The most important thing to see from this graph is that $O = 8$ is the best parameter across all machine sizes except for the largest we looked at (where the best value of O is 4).

While these figures may differ depending on network volume and other factors, we do expect NIFDY performance to stay constant or increase as the network size grows—while keeping the same small fixed parameters in NIFDY. In fact, the results should be even more favorable on networks in which the bisection bandwidth does not scale linearly with the number of nodes, such as a two-dimensional mesh. In these cases the per-node bandwidth would have to decrease as the machine size grows in order to avoid congestion at the bisection, making smaller values of O and B more desirable.

4.3 Cyclic Shift

In this subsection we consider a specific traffic pattern, the cyclic shift (C-shift) studied in [BK94], which provides all-to-all communication. We implemented this traffic pattern using the "real traffic" interface to our simulator and the CM-5–style network in order to make comparisons with [BK94].

The C-shift communication pattern consists of $P-1$ phases. In the first phase, processor i sends to processor $(i+1)$ mod P; in phase p, processor i sends to processor $(i+p)$ mod P; until $p = P-1$. As long as the phases remain separate, each receiver is matched with exactly one sender. However, as observed in [BK94], some nodes may finish the current phase early and move to the next phase, resulting in one node receiving from two senders. This slows the progress of both senders, allowing other senders to catch up and aggravating the condition. Figure 5 shows the number of packets in the network for each receiver as the pattern progresses, clearly indicating the accumulation of packets outside certain receivers. One solution used in Strata [BK94] is to insert global barriers between phases.

Results are summarized in Figure 6. Using NIFDY's congestion control alone results in better performance than optimized barriers. When NIFDY's in-order delivery is exploited, the benefit is even greater. These results can be explained by looking at Figure 5. Some piling up does occur with NIFDY(due to the different path lengths between different pairs of nodes), but these perturbations dissipate and the network returns to even utilization of all receivers. This dissipation occurs because the "rightful" sender to a receiver has the advantage that it owns the bulk dialog to that receiver. Thus it will be allowed to finish rapidly and move on to the next receiver; at that point the sender behind it can attain the bulk dialog. Although this effect is dependent on NIFDY parameters and network characteristics, in all cases performance was much better with NIFDY than with nothing at all.

4.4 EM3D

EM3D, a program for solving electromagnetic problems in three dimensions and a common parallel benchmark [CDG+93], was also used to drive our simulations. The results of our simulations for a number of different networks are summarized in Figure 7 (for the light network load) and Figure 8 (for the heavy network load). For networks that deliver packets out of order, two NIFDY results are presented: one which gives the benefit just from the flow control ("NIFDY–"), and another in which the Split-C library that interfaced to our network simulator was altered to take advantage of the in-order delivery provided by NIFDY. For networks that deliver packets

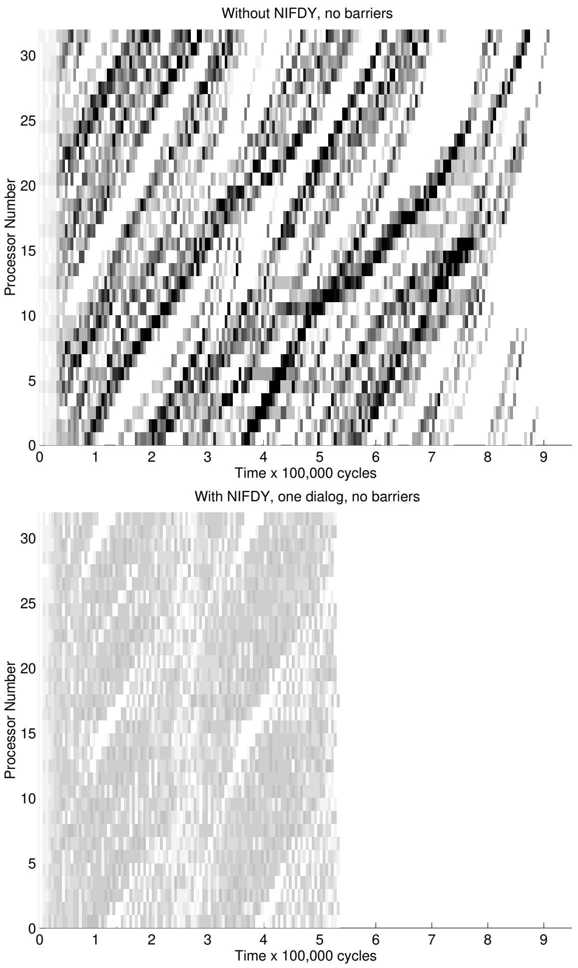

Figure 5: *Network congestion with C-shift: pending packets per receiver without and with* NIFDY *(no barriers in either case). Shading is interpolated between white for no pending packets and black for 20 or more pending packets. In both cases, the same number of packets are transferred, but* NIFDY *finishes earlier.*

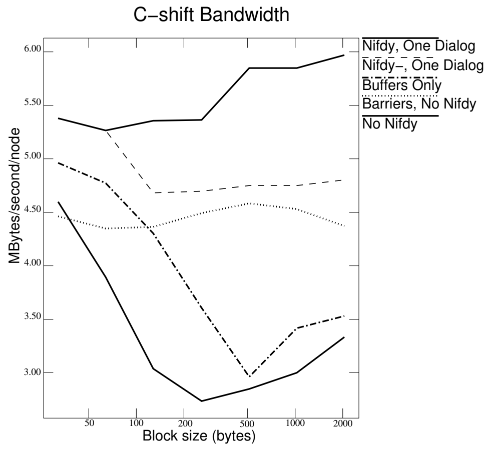

Figure 6: *Throughput for C-shift on 32-node CM-5 network.*

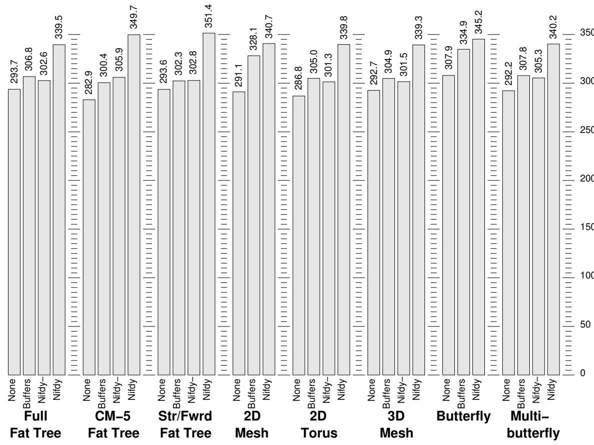

Figure 7: *EM3D iterations per second with less communication. (In the computation graph generated by the parameters, most arcs are local to processors.) n_nodes = 200, d_nodes = 10, local_p = 80, dist_span = 5.* NIFDY*– reflects benefit from flow control only;* NIFDY *exploits in-order delivery as well.*

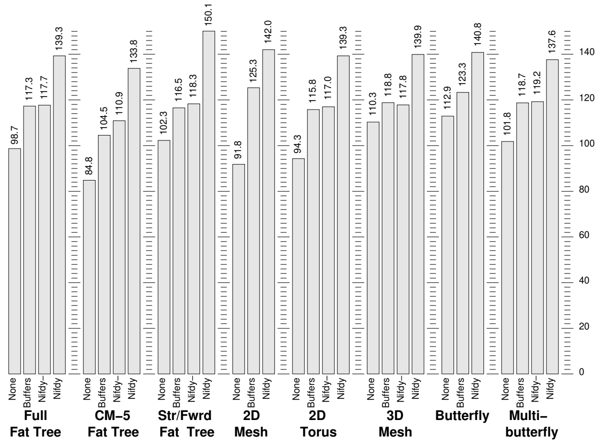

Figure 8: *EM3D iterations per second with more communication. (In the computation graph generated by the parameters, most arcs are between processors.) n_nodes = 100, d_nodes = 20, local_p = 3, dist_span = 20.* NIFDY*– reflects benefit from flow control only;* NIFDY *exploits in-order delivery as well.*

in order (the 2D mesh and the butterfly), the library intended for in-order delivery was used for all runs.

Without in-order delivery, the difference between NIFDY and the buffers-only configurations is negligible. Once the library takes advantage of the in-order delivery provided by NIFDY, it outperforms the buffers-only configuration in all cases.

4.5 Radix Sort

Finally, we ran simulations of a radix sort based on [Dus94]. Each iteration of radix sort consists of two communication phases: scan and coalesce. In the scan phase, a scan addition is performed across all processors for each bucket; this involves nearest-neighbor communication. The most notable feature of this is that the overall communication phase runs faster if delays are inserted between

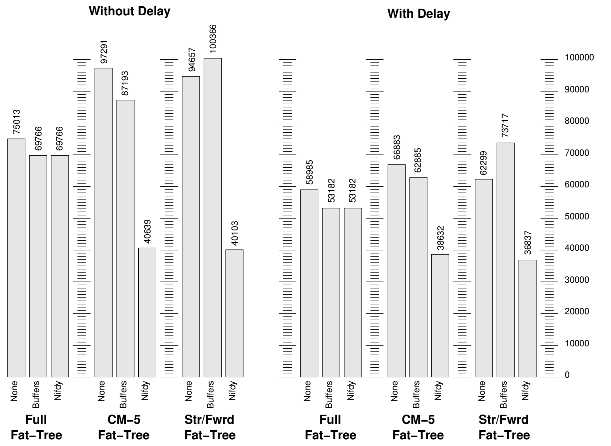

Figure 9: *Cycles for one scan phase of radix sort. "With Delay" indicates that artificial delays are inserted between consecutive sends to the same destination.*

successive sends. Without delays, the sends from one processor cause the next processor in the pipeline to continually receive with no chance to send, serializing the entire scan. We studied versions both with and without the delay. In the coalesce operation, the keys are sent to the appropriate destination using one message for each key; assuming a random initial key distribution, the one-packet messages are sent to a random sequence of destination processors.

Figure 9 shows results for the scan phase using an 8-bit radix on 64 processors. While adding delays between successive sends helped in all cases, it was more critical when NIFDY was not included. When NIFDY is included, its protocol causes the sender to slow down; this allows all the processors to continue to send as well as receive. Networks with higher latencies, e.g., the store and forward fat tree, get a bigger gain from NIFDY, than those with lower latencies, like the full fat tree. This exemplifies what we found in many cases: the locally restrictive NIFDY protocol actually results in *more* global throughput.

Results for the coalesce phase (not shown) were virtually identical with and without NIFDY. There was not enough congestion for NIFDY's flow control to help, and with this algorithm in-order delivery is not beneficial. On the other hand, NIFDY's restrictiveness did not hurt performance.

4.6 Discussion

NIFDY performs well for three real communication patterns which form the basis for many parallel programs. The NIFDY protocol may seem restrictive, but NIFDY's admission control reduces congestion in the network. Our results show that it delivers more packets than the same network without NIFDY, and roughly the same as when NIFDY's buffering is used without the protocol. When NIFDY's in-order delivery is taken into account, NIFDY is seen to give a clear benefit for all shallow networks.

NIFDY helps different networks in different ways. For networks with a single path between each sender and receiver, packets are already delivered in-order, so NIFDY gives no additional benefit in that respect; however, such networks degrade rapidly in the presence of congestion, which NIFDY helps avoid. For networks with multiple routes between each sender and receiver, performance does not degrade as rapidly because packets can travel alternate routes around hot spots. In such cases, NIFDY's main benefit is reordering the packets that otherwise would arrive out of order.

Finally, we saw that on most networks many of the communication patterns can be sped up by carefully crafted software techniques. Without either the techniques or NIFDY, many of these patterns ran poorly. With NIFDY, intelligent software techniques were useful, but they were not as important. In general, NIFDY provides a safety net when software network management techniques are not or cannot be applied.

5 Related Work

Flow control (FC) and congestion control (CC) have received much attention in LAN and WAN research. The larger packets and longer latencies in these types of network make software implementation of FC and CC protocols practical. Our method, being simple enough to implement in hardware, provides FC and CC for the type of low-latency, high-bandwidth networks where software protocols are not practical. Specifically, NIFDY does not require any intelligence within the network switches and it does not require nodes to keep per-receiver connection or credit state.

In multiprocessor networks, the need to reduce software communication overhead while making good use of network bandwidth has inspired many attempts to "raise the functionality of the network"—usually by reducing congestion or providing in-order packet delivery. Most of these projects have taken a different approach from NIFDY's: they have added functionality to the network routers rather than just to the network interfaces.

The METRO router [DCB+94] provides in-order delivery while taking advantage of random wiring in expansion networks. The router is a dilated crossbar and is used as a building block for indirect expander networks such as multibutterflies and metabutterflies [CBLK94]. A sender attempts to make a connection randomly through successive dilated crossbars; if a connection attempt is blocked, the path is torn down, and the connection is retried later. Once a connection is established, it remains fixed, and thus transfers are in order. The cost of blocked connection attempts means that METRO must make sure that most connection attempts succeed; thus it is important to have large bandwidth throughout the network, probably much more than is needed to carry the average load. NIFDY allows network utilization closer to its theoretical maximum, while preventing the user from pushing the network out of its operating range. While METRO requires nontrivial intelligence at the transfer endpoints, its key characteristics arise from its router design. NIFDY, in contrast, can be used with a variety of networks.

Compressionless Routing (CR) [KLC94] also provides in-order delivery. CR, which relies on wormhole routing, pads packets with enough space to ensure that pushing the entire packet onto the network implies that the head of the packet has already entered the destination, at which point the packet is guaranteed to be completely consumed. If the packet cannot be pushed out within a preset amount of time, the transmission is aborted and the flits already in the network are killed. Abstractly, there are some similarities between CR and the basic NIFDY protocol. With both there can be at most one unreceived packet in the network between any source/destination pair. In addition, CR also uses an ack, albeit an implicit one—lack of backpressure—which travels from the destination to the source on the switch-level ack control wires. However, our implementation differs markedly. The explicit acks in NIFDY consume some bandwidth, but there is no need to add wasteful padding to short packets. The tearing down of packets due to blockage causes instability at high network load: the average amount of bandwidth consumed per successful transfer increases, making the congestion worse. In contrast, our method performs best under high load and prevents the network from being pushed into a regime of declining throughput. NIFDY is fairly insensitive to our preset parameters; with CR, a poorly chosen timeout period may drastically affect performance (although our extension to NIFDY for handling dropped packets will have the same sensitivity in this respect). NIFDY is very general since it is logically separate from the network; it can be used with wormhole, cut-through, or store-and-forward routing, and can be added to an existing network with no change to the network itself. In contrast, CR can be used only with wormhole routing, and it requires the network routers and interfaces to support the killing of packets. However, CR, unlike NIFDY, can be used with networks that are not deadlock-free.

Finally, there are many software techniques that can be used to reduce network congestion [BK94]. These techniques, such as structuring communication as series of permutations allowing one-on-one transfers, are beneficial even with NIFDY. However, NIFDY will add robustness to the system and be especially effective with traffic patterns that are difficult for software to manage, in particular those with no global structure. In some cases the behavior of NIFDY with irregular traffic will mimic the software techniques used with regular communication. For example, NIFDY automatically interleaves packets to different destinations. And NIFDY effectively implements bandwidth matching—injecting packets at the rate at which it receives acks, which is the rate at which the receiving processor is pulling packets out of the network. NIFDY also handles the more general case with multiple nodes sending to one receiver, returning acks only at the rate at which the receiver accepts packets. This throttles the combined injection rate of all the senders to a level that the receiver can handle. It would be difficult and expensive to implement such dynamic bandwidth matching in software.

6 Future Work

6.1 Changes to ack strategy

There are two changes to the current protocol that we would like to study: allowing acks to be combined with reply messages, which should reduce network traffic; and allowing packets that don't require acks.

In the protocols described in this paper, the ack packets are always generated by NIFDY. In many situations the user code will also send a reply message to the source processor. Instead of sending both a NIFDY-generated ack and a user reply we could piggyback the ack in the reply. This seems to be a good idea, since if the sender is waiting for a reply it probably won't have any other packets for the destination processor until the reply is received. Adding this protocol requires only an additional bit in the header and a comparator in the "to processor" block in Figure 1.

NIFDY could be configured so that the processor indicates when it wants to bypass the NIFDY protocol. This could be done when the processor does not care about in-order delivery and knows the that packets will not contribute to congestion (or is willing to take the risk). Such packets would be eligible to be sent immediately, and would be handled just like scalar packets at the receiver except that no acknowledgements would be sent. This type of traffic could co-exist with traffic obeying the NIFDY protocol.

6.2 Networks of Workstations

We have shown how NIFDY increases performance of reliable networks for MPPs. For shallower networks scalar packets combined with a medium-sized OPT were shown to be sufficient due to the small round-trip latency. For deeper networks, we added bulk dialogs to overcome this latency. Here we extend NIFDY for networks of workstations which may drop packets. Our goal is to make the network transparent to the application and for it to be scalable.

To handle networks that drop packets the sender must be able to retransmit packets. In addition, the receiver must be able to distinguish and eliminate duplicate packets. To accomplish retransmission we add one timer and one message buffer per entry in the OPT and per outgoing bulk packet. The outgoing packet is copied into the buffer and the timer is set when the packet is sent. If an ack is received before the timer expires, the timer is reset and the buffer is freed for future use. If no ack is received before the timer goes off, the packet is retransmitted. To distinguish duplicate packets, one additional bit in the header is enough for both scalar and bulk packets.

This simple extension—a bit in each header and some additional state and buffering on each NI—allows NIFDY to hide the implementation details of the network from system software and user applications alike. We have used simple hardware to mask an exceptional condition (viz., the dropping of a packet), which should reduce software overhead at both the sender and the receiver. According to [KC94], this should reduce the cost of sending and receiving messages by 30% to 50%.

6.3 Further Experiments

In addition to the extensions proposed above, we believe that we have just begun to understand how the network parameters affect the throughput and latency of messages on the network. While we have a good understanding of the O and P parameters and how they interact with traffic patterns, we have yet to study the interaction among transfer lengths, W, and the optimal point for requesting a bulk dialog.

We also plan to extend the simulator to study how NIFDY interacts with adaptive routing on a mesh, which in the past has not performed well enough to justify its expense. Adding the admission control and in-order delivery of NIFDY may help adaptive routing reach its potential.

7 Summary and Conclusion

In this paper we have proposed a network interface, NIFDY, which increases network performance and decreases software overhead without restricting routing choices in the network. We have shown that it is possible to achieve these goals simultaneously by adding modest resources only at the network interface and without having to push any functionality throughout the network.

In essence, the basic NIFDY protocol is an optimized credit-based scheme where every sender implicitly has one credit for each receiver. Because senders record only receivers with zero credits rather than maintaining state for all receivers, the resources consumed at each sender scale with the number of outstanding packets rather than the total number of nodes. Because credits are good only for a particular processor, the protocol can easily adapt to bimodal MPP traffic.

We built a general-purpose simulator to test these ideas. We verified the simulator against a real machine, the CM-5, and then used the simulator to evaluate the performance advantage of a NIFDY network interface attached to a variety of network fabrics, including meshes, tori, butterflies, and fat trees. We showed that on every network, and all synthetic and real traffic patterns, NIFDY increased packet throughput to a level comparable to that of having added more buffers. In addition, since NIFDY delivers packets in order, it increased total payload delivered on all networks. For real traffic patterns we saw increases from 10% (under light loads presented in EM3D) to as much as 100% (under all-to-all transfers) over just having added more buffers.

Using the simulator we also showed that the resources needed by NIFDY are constant (or decreasing) with respect to the number of nodes in the network. In particular, for all the networks studied, an outstanding packet table of size 8 combined with a packet pool of 16 and a single bulk dialog with a window of 8 were more than enough resources for even large machines. In fact, on most networks fewer resources than these gave better results. Thus, given the performance advantages of NIFDY, the small additional chip area needed over plain network interfaces is a worthwhile investment.

Acknowledgments

We are grateful to the anonymous referees for their valuable comments. We would also like to thank Krste Asanović, Eric Brewer, David Culler, Andrea Dusseau, Steve Lumetta, Klaus Erik Schauser, Nathan Tawil, and John Wawrzynek for their comments on earlier versions of this paper, and Su-Lin Wu for her contributions to early stages of this work. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship. Timothy Callahan received support from an NSF Graduate Fellowship and ONR Grant N00014-92-J-1617. This work also received support through NSF Presidential Faculty Fellowship CCR-92-53705 and LLNL Grant LLL-B283537-Culler.

References

[Aga91] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, vol.2(no.4):398–412, Oct. 1991.

[BK94] E.A. Brewer and B.C. Kuszmaul. How to get good performance from the CM-5 data network. In *Proceedings Eighth International Parallel Processing Symposium*, pages 858–67. IEEE Comput. Soc. Press, 1994.

[BT89] R.G. Bubenik and J.S. Turner. Performance of a broadcast packet switch. *IEEE Transactions on Communications*, vol.37(no.1):60–9, Jan. 1989.

[CBLK94] F.T. Chong, E.A. Brewer, F.T. Leighton, and T.F. Knight, Jr. Building a better butterfly: The Multiplexed Multibutterfly. In *Proc. International Symposium on Parallel Architectures, Algorithms, and Networks*, Kanazawa, Japan, December 1994.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proc. Supercomputing '93*, Portland, Oregon, November 1993.

[Cul94] David E. Culler. Multithreading: Fundamental limits, potential gains, and alternatives. In R.A. Iannuci, G.R. Gao, Jr. Halstead, R.H., and B. Smith, editors, *Multithreaded Computer Architecture*, chapter 6, pages 97–138. Kluwer Academic Publishers, 1994.

[Dal90] W.J. Dally. Virtual-channel flow control. In *Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 60–8. IEEE Comput. Soc. Press, 1990.

[Dal91] W.J. Dally. Express cubes: improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, vol.40(no.9):1016–23, Sept. 1991.

[DCB+94] A. DeHon, F. Chong, M. Becker, E. Egozy, H. Minsky, S. Peretz, and Jr. Knight, T.F. Metro: a router architecture for high-performance, short-haul routing networks. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 266–77. IEEE Comput. Soc. Press, 1994.

[Dus94] Andrea Carol Dusseau. Modeling parallel sorts with LogP on the CM-5. Technical Report UCB//CSD-94-829, University of California at Berkeley, May 1994.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Computer Communication Review*, pages 314–29, Aug. 1988.

[Jai90] R. Jain. Congestion control in computer networks: issues and trends. *IEEE Network*, vol.4(no.3):24–30, May 1990.

[KC94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: Where does the time go? In *Proc. of 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[KLC94] J.H. Kim, Ziqiang Liu, and A.A. Chien. Compressionless routing: a framework for adaptive and fault-tolerant routing. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 289–300. IEEE Comput. Soc. Press, 1994.

[KMCL93] H.T. Kung, Robert Morris, Thomas Chaaruhas, and Dong Lin. Use of link-by-link flow control in maximizing atm networks performance: Simulation results. In *Proceedings IEEE Hot Interconnects Symposium '93*, August 1993.

[KS91] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In *Computer Architecture News*, pages 212–21, May 1991.

[LAD+92] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynmann, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, Shaw-Wen Yang, and R. Zak. The network architecture of the connection machine CM-5. In *SPAA '92. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–85. ACM, 1992.

[Mar] Richard Martin. Personal Communication.

[RJ90] K.K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems*, vol.8(no.2):158–81, May 1990.

[SBB+91] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, and C.P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, vol.9(no.8):1318–35, Oct. 1991.

[SS89] S.L. Scott and G.S. Sohi. Using feedback to control tree saturation in multistage interconnection networks. In *16th Annual International Symposium on Computer Architecture*, pages 167–76. IEEE Comput. Soc. Press, 1989.

[vE93] Thorsten von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* PhD thesis, University of California at Berkeley, December 1993.