Self-Tuned Congestion Control for Multiprocessor Networks

Mithuna Thottethodi[†] Alvin R. Lebeck[†] Shubhendu S. Mukherjee[‡]

[†]Department of Computer Science,
Duke University,
Durham, NC 27708-0129.
{mithuna,alvy}@cs.duke.edu

[‡]VSSAD, Alpha Development Group,
Compaq Computer Corporation,
Shrewsbury, MA.
shubu.mukherjee@compaq.com

Abstract

Network performance in tightly-coupled multiprocessors typically degrades rapidly beyond network saturation. Consequently, designers must keep a network below its saturation point by reducing the load on the network. Congestion control via source throttling—a common technique to reduce the network load—prevents new packets from entering the network in the presence of congestion. Unfortunately, prior schemes to implement source throttling either lack vital global information about the network to make the correct decision (whether to throttle or not) or depend on specific network parameters, network topology, or communication patterns.

This paper presents a global-knowledge-based, self-tuned, congestion control technique that prevents saturation at high loads across different network configurations and communication patterns. Our design is composed of two key components. First, we use global information about a network to obtain a timely estimate of network congestion. We compare this estimate to a threshold value to determine when to throttle packet injection. The second component is a self-tuning mechanism that automatically determines appropriate threshold values based on throughput feedback. A combination of these two techniques provides high performance under heavy load, does not penalize performance under light load, and gracefully adapts to changes in communication patterns.

1 Introduction

Tightly-coupled multiprocessors provide the performance and ease of programming necessary for many commercial and scientific applications. Their interconnection networks provide the low latency and high bandwidth communication required for a variety of workloads. The advent of multiprocessor systems built with highly aggressive, out-of-order, and speculative microprocessors, simultaneous multithreaded processors [9], and chip multiprocessors [7], promises to dramatically increase the offered load on such multiprocessor networks. Unfortunately, most multiprocessor networks suffer from tree saturation under heavy load [22] and could become a key performance bottleneck.

Tree saturation occurs when multiple packets contend for a single resource (e.g., a link between nodes) creating a hotspot. Since only one packet can use the resource, other packets must wait. These waiting packets occupy buffers and thus delay other packets, even though they may be destined for a completely different node and share only one link on their paths to their respective destinations. This process continues, waiting packets delay other packets producing a tree of waiting packets that fans out from the original hotspot.

The performance degradation caused by network saturation can be severe, as illustrated in Figure 1. The y-axis corresponds to delivered bandwidth (flits/node/cycle) while the x-axis shows offered load in terms of packet injection rate (packets/node/cycle). The two lines correspond to different communication patterns: randomly selecting a destination node (*random*), and using the node number with its most significant and least-significant bits switched as the destination (*butterfly*).

From Figure 1 we can make two important observations. First, both communication patterns incur dramatic reductions in throughput when the network reaches saturation. The second observation is that the network saturates at different points for the different communication patterns.

One way to prevent network saturation is to use source throttling, which prevents source node packet injection when congestion is detected. An oracle could achieve this by knowing both the communication pattern and the packet injection rate that maximizes performance. The challenge is to develop a realistic implementation that can prevent network saturation and adapt to variations in communication

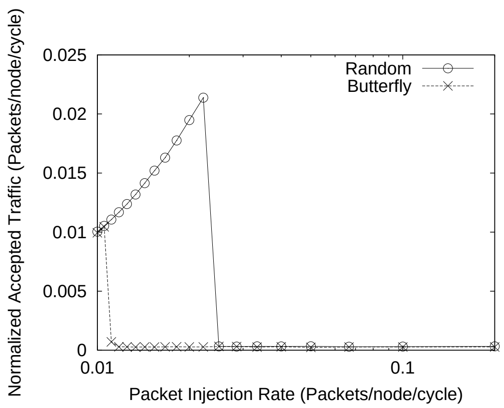


Figure 1. Performance Breakdown at Network Saturation, 16x16 2D network, adaptive routing, deadlock recovery

patterns.

This paper presents a self-tuned source throttling mechanism for multiprocessor interconnection networks. Our solution is comprised of two key components: a technique to obtain global knowledge of network state and a self-tuning mechanism to automatically determine when network saturation occurs.

We use global knowledge of the number of full network buffers to estimate network congestion. Global information allows us to detect congestion earlier than alternative approaches that wait for network backpressure to create locally observable indicators of congestion (e.g., local buffer occupancy, timeouts). The global estimate is compared against a threshold to control packet injection. If the estimate is higher than the threshold, packet injection is stopped. When the estimate drops below the threshold, packet injection is resumed.

The second key aspect of our source throttling implementation is a self-tuning mechanism that monitors network throughput and automatically determines the appropriate threshold value. This eliminates manual tuning and allows our scheme to adjust to variations in communication patterns.

We believe that our congestion control mechanism is generally applicable to a broad range of packet-switched, multiprocessor networks, including virtual cut-through [15] networks and wormhole networks [6, 5]. However, in this paper we evaluate the technique in the context of wormhole switched, k-ary,n-cube networks.

Simulation results for a 16-ary,2-cube (256 node network) show that our congestion control technique prevents the severe performance degradation caused by network saturation. By limiting packet injection, our scheme sustains high throughput and low latency. Compared to an alternative approach that uses local estimates of congestion [2], our scheme is superior because global congestion estimation enables our technique to detect congestion in its early stages. We also show that a single static threshold cannot accommodate all communication patterns because a single threshold overthrottles some workloads and does not prevent saturation in other ones. In contrast, simulations reveal that our self-tuning technique automatically adapts to various communication patterns, including bursts of different patterns.

The remainder of this paper is organized as follows. Section 2 provides background information and discusses related work. Section Section 3 and Section Section 4 discuss the two key innovations of this paper. Section 3 presents our proposed global information gathering scheme. Section 4 describes our self-tuned congestion control scheme. Section 5 presents our experimental methodology and simulation results. Section 6 summarizes this paper.

2 Background and Related Work

High performance interconnection networks in tightly coupled multiprocessors can be achieved by using wormhole [6, 5] or cut-through switching [15], adaptive routing [12], and multiple virtual channels [4]. The Cray T3E [25] and SGI Origin [18] machines use a combination of these techniques for their interconnection networks. In these systems communication occurs by sending packets of information that are routed independently through the network. Each packet is composed of flits (flow control units) that are transferred between network nodes.[1]

Both wormhole routing and cut-through switching can suffer from network saturation. In wormhole switching, when a node receives the header flit (which typically contains the routing information), it immediately selects a route and forwards the flit to the next node. This can provide very low latency compared to store-and-forward routing where the entire packet is received by a node before forwarding it. However, when a packet blocks in a wormhole network, its flits occupy buffer space across several network nodes, and can exacerbate tree saturation. In contrast, routers using cut-through switching buffer blocked packets in the router itself. Nevertheless, even cut-through switching can suffer from tree saturation when the router buffers fill up.

Adaptive routing dynamically chooses from multiple potential routes based on current local network state. This can help alleviate the effects of tree saturation experienced by deterministic routing algorithms under heavy load, and thus provide higher performance. Unfortunately, full adaptive routing can cause potential deadlock cycles, which can exacerbate network saturation. While adaptive routing helps

[1]For ease of exposition we assume each network node contains a processor, memory and a network router.

alleviate light to moderate congestion, in this paper we focus on source-throttling as a congestion control technique to prevent network saturation.

Virtual channels allow multiple packets to share a single physical link, thus reducing the effects of tree saturation, and can be used to eliminate deadlocks. Deadlock avoidance schemes work by preventing the cyclic dependencies between storage resources. In particular, we consider a scheme that reserves a small set of virtual channels for deadlock-free routing [8], while the remaining virtual channels use fully adaptive routing. This technique guarantees forward progress, since packets routed over the special channels will never deadlock, and eventually free up resources for the fully adaptive channels.

Deadlock recovery [17] is an alternative to deadlock avoidance that can potentially achieve higher performance. Deadlock recovery uses full adaptive routing on all virtual channels, detects when deadlocks occur (typically via timeouts), then recovers by routing packets on a deadlock free path which uses a central per-node buffer. This approach can also be imagined as containing two virtual networks: one can suffer deadlock and the other is guaranteed to be deadlock-free. The main difference is that this approach uses a per-node buffer whereas the deadlock avoidance scheme requires buffers per physical channel.

In either deadlock avoidance or recovery, the frequency of deadlocks in the adaptive channels increases dramatically when the network reaches saturation [29]. When this occurs, packets are delivered over the relatively limited escape bandwidth available on the deadlock-free paths. This causes a sudden, severe drop in throughput and corresponding increase in packet latency. Therefore, it is crucial to avoid network saturation in these systems.

To keep the network below saturation and avoid the resulting performance degradation, it is necessary to implement a congestion control mechanism. This mechanism should be self-tuning, thus eliminating the need for a system designer, administrator, or application programmer to tune various parameters and allowing the system to adapt to changes in communication patterns, load levels, and network topologies. Such a self-tuned system may require timely information about the global network state to correctly tune the congestion control algorithm.

The remainder of this section examines prior congestion control mechanisms for tightly coupled multiprocessors in the context of these desirable properties. We also examine congestion control mechanisms used in LANs/WANs and discuss the applicability of those techniques to tightly coupled multiprocessors.

2.1 Related Work

Most previous work on congestion control for multiprocessor networks relies on estimating network congestion independently at each node and limiting packet injection when the network is predicted to be near saturation. This reduces the problem to finding a local heuristic at each node to estimate network congestion. Lopez et al. [19, 20] use the number of busy output virtual channels in a node to estimate congestion. Baydal et al. [2] propose an approach that counts a subset (free and useful) of virtual channel buffers to decide whether to throttle or not. Because the above schemes rely on local symptoms of congestion, they lack knowledge about the global network state, and are unable to take corrective action in a timely manner. This reduces their effectiveness under different network load levels and communication patterns.

Smai and Thorelli describe a form of global congestion control [26]. A node that detects congestion (based on time-outs) signals all the other nodes in the network to also limit packet injection. This approach requires tuning the appropriate time-outs, and when the timeouts are tuned for robustness at higher loads, there is a performance penalty for light loads. Scott and Sohi describe the use of explicit feedback to inform nodes when tree-saturation is imminent in multistage interconnection networks [24]. This approach also requires tuning of thresholds.

The technique proposed by Kim et al. [16] allows the sender to kill any packet that has experienced more delays than a threshold. This approach pads shorter packets to ensure that the sender can kill a packet at any time before its first flit reaches the destination. This can cause larger overheads when short messages are sent to distant nodes.

The above techniques for congestion control in multiprocessor networks all attempt to prevent network saturation at heavy loads. Unfortunately, these techniques either require tuning, lack necessary information about a network's global state to take preventive actions in a timely fashion, or do not provide high performance under all traffic patterns and offered load levels.

Flit-reservation flow control is an alternative flow control technique which improves the network utilization at which saturation occurs [21]. It uses control flits to schedule bandwidth and buffers ahead of the arrival of data-flits. This prescheduling results in better re-use of buffers than waiting for feedback from neighboring nodes to free up buffers. Basak and Panda demonstrate that consumption channels can be a bottleneck that can exacerbate tree saturation. They show that saturation bandwidth can be increased by having an appropriate number of consumption channels [1].

LANs (Local Area Networks) and WANs (Wide Area Networks) use self-tuned congestion control techniques. Various flavors of self-tuning, end-to-end congestion avoidance and control techniques have been used in the TCP protocol [13, 3]. TCP's congestion control mechanism uses time-outs and dropped/unacknowledged packets to locally estimate global congestion and throughput. If congestion is detected, the size of the sliding window, which controls the number of unacknowledged packets that can be in flight, is reduced. Floyd and Jacobson [11] proposed a scheme where TCP packets are dropped when a router feels that congestion is imminent. Dropped packets give an early indication to end hosts to take corrective action and scale back offered load. Floyd [10] also proposed a modification of TCP where "choke packets" are explicitly sent to other hosts. Ramakrishnan and Jain describe a similar mechanism for DECbit to explicitly notify congestion whereby gateways set the ECN (*Explicit Congestion Notification*) bit depending on average queue size [23].

Congestion control in ATM [14] uses explicit packets called *Resource Management (RM) cells* to propagate congestion information. Switches along the packet path modify bits in the RM cells to indicate the highest data rate they can handle. The end-hosts are limited to using the maximum data-rate indicated by the switches to not overwhelm the network and/or switches.

The above congestion control mechanisms for LANs and WANs are not directly applicable in multiprocessor networks. LANs and WANs can drop packets because higher network layers will retransmit dropped packets for reliable communication. The dropped packets serve as implicit hints of network congestion. However, multiprocessor networks are typically expected to guarantee reliable communication. Thus, additional complexity would have to be built-in to store and retransmit dropped packets. The alternative idea of propagating congestion information explicitly can be used.

The challenge is in determining the appropriate set of mechanisms and policies required to provide a self-tuned congestion control implementation for preventing saturation in multiprocessor networks. In this paper, we present our solution for regular interconnection networks with adaptive routing, wormhole switching, and either deadlock recovery or deadlock avoidance.

Our solution is based on two key innovations that collectively overcome the limitations of previous congestion control techniques. First, we use a *global knowledge based congestion estimation* that enables a more timely estimate of network congestion. The second component is a *self-tuning* mechanism that automatically determines when saturation occurs allowing us to throttle packet injection. The next two sections elaborate on each of these key components.

3 Global Congestion Estimation

Any congestion control implementation requires a timely way to detect network congestion. Previous techniques estimate network congestion using a locally observable quantity (e.g., local virtual buffer occupancy, packet timeouts). While these estimates are correlated to network congestion, we claim that waiting for local symptoms of network congestion is less useful primarily because, by that time, the network is already overloaded.

Consider the case when network congestion develops at some distance from a given node. Schemes that use local heuristics to estimate congestion rely on back-pressure to propagate symptoms of congestion to the node (e.g. filling up of buffers, increase in queue delays, etc.). The node takes no corrective action until congestion symptoms are observed locally.

It is possible to detect network congestion in its early stages by taking global conditions into account. To achieve this, we use the fraction of full virtual channel buffers of all nodes in the network as our metric to estimate network congestion. This ensures that far away congestion is accounted for early enough to take corrective action. However, there is additional cost, both hardware and latency, to propagate the global information.

Our scheme counts full buffers to estimate congestion but does not take the distribution of these full buffers among the nodes into account. At first glance, this appears to be a serious limitation because our scheme is unable to distinguish between a case with localized congestion (i.e., a large fraction of full buffers are in relatively few nodes in the network) and a benign case (in which the same number of full buffers are distributed more or less evenly among all the nodes of the network). But the adaptivity of our self-tuning mechanism reduces the impact of this problem by setting the threshold differently in the two cases. Our mechanism will set a higher threshold for the benign case than for the case with localized congestion.

In the next section, we show how global information can be gathered with reasonable cost and used to achieve a robust, self-tuned congestion control implementation.

3.1 Implementing Global Information Gather

Our technique requires that every node in the network be aware of the aggregate number of full buffers and throughput for the entire network. (We explain the relationship between full buffers, offered load and delivered bandwidth in Section 4.) There are a variety of ways to implement this all-to-all communication. In this section we discuss three alternatives: piggy-backing, meta-packets, and a dedicated side-band.

One approach to distribute information in the network is to piggy-back the extra information on normal packets. This approach has the disadvantage that it is difficult to guarantee all-to-all communication. Since only nodes involved in communication see the piggy-backed information, it is possible that some nodes will not see the information for an extended period of time, if at all. This can reduce the accuracy of the congestion estimate, thus reducing the effectiveness of the congestion control scheme.

An alternative approach is to send out special meta-packets containing the congestion and throughput information. The required all-to-all communication can be guaranteed by this approach. However, guaranteeing delay bounds may involve additional complexity. Meta packets flooding the network will also consume some of the bandwidth and may add to the congestion. Adding a high-priority virtual channel reserved for these meta-packets may be one way of addressing these concerns.

For this paper, we use an exclusive side-band reserved for communicating the congestion and throughput information. This is the costliest implementation in terms of additional hardware and complexity. However, it is easy to guarantee delay bounds on all-to-all communication and it does not affect performance of the main data network. While the extra bandwidth available on the side-band could be used for general packet routing, it will only postpone network saturation for a short-time, and not provide a complete solution like our congestion control scheme. From the above discussion, we see several issues surrounding the global information gathering mechanism. Our future work includes a more thorough examination of these issues and implementations.

The side-band incurs a neighbor-to-neighbor communication delay of h cycles. We use a *dimension-wise aggregation* scheme. Every node communicates its number of full buffers and throughput in both directions along the lowest dimension of the network. Each node that receives such information computes the aggregate and has the aggregate information for all its neighbors along the zeroth dimension at the end of $k/2$ hops or $(k/2) * h$ cycles. The nodes then communicate the aggregates to neighbors along the next higher dimension. Continuing this procedure along every dimension, global aggregation in a full-duplex, k-ary,n-cube network completes in $(k/2) * h * n$ cycles. Assuming $h = 2$, for our network configuration $(n = 2, k = 16)$ it takes 32 cycles. We refer to the time for such an all-to-all communication as the *gather-duration* (g).

The mechanism described above provides g-cycle delayed snapshots of the network congestion every g cycles. Our congestion control policy requires us to compare, in every cycle, the current estimated congestion to the threshold. If we're currently at time t and we have observed previous network snapshots at S_2, S_1, S_0 and so on, we must estimate the network congestion at time t based on the previous snap-shots of global network congestion.

The simplest solution is to use the state observed in the immediately previous network snapshot until the next snapshot becomes available. We use a slightly more sophisticated method to estimate network congestion that computes a linear extrapolation based on the previous two network-snapshots. In general, any prediction mechanism based on previously observed network states can be used to predict network congestion. We leave evaluation of alternative prediction techniques for future work. On average, we found the linear extrapolation technique yields an improvement in throughput of 3% for the deadlock avoidance configuration and 5% for the deadlock recovery configuration.

To communicate congestion information, nodes exchange full buffer counts. The number of bits needed to represent this information depends on the number of buffers in the network. We specify the network configuration we use and the number of bits needed to represent congestion information for that configuration in Section 5.

In summary, global measurement of virtual buffer occupancy provides an early estimate of network congestion. This estimate is compared against a threshold to determine if packet injection should stop or resume. Obtaining information on the global state of the network is only part of the solution. To translate this congestion estimate to good congestion control, we have to properly choose the threshold. Our self-tuning mechanism, described in the next section, dynamically tunes the threshold to the appropriate values.

4 A Self-Tuning Mechanism

Proper threshold selection is a crucial component of our congestion control implementation. Inappropriate threshold values can produce unstable behavior at high loads or unnecessarily limit performance for light loads. Furthermore, there is no single threshold that works well for all communication patterns. This section presents a technique to automatically determine the proper threshold value.

The goal of our self-tuning mechanism is to maximize delivered throughput without dramatic increases in packet latency. Therefore, we can view our task as an optimization problem with delivered bandwidth as an objective function dependent on the number of full virtual buffers. Consider the relationship between offered load, full buffers and delivered bandwidth (See Figure 2). As offered load increases from zero, the number of full virtual buffers and delivered bandwidth also increase. When saturation occurs, the delivered bandwidth decreases while the number of full virtual buffers continues to increase.

Our self-tuning technique is attempting to find the number of full virtual buffers (i.e., the threshold value) that maximizes delivered throughput (B in Figure 2). To achieve this, we use a hill-climbing algorithm including a technique to

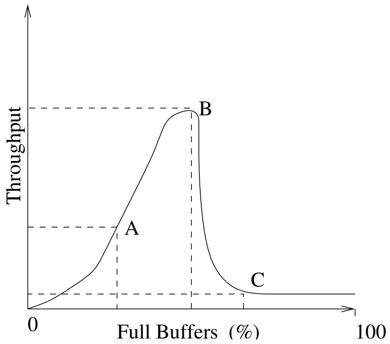


Figure 2. Throughput vs. Full Buffers

Drop in Bandwidth $> 25\%$?	Currently Throttling?	
	Yes	No
Yes	Decrement	Decrement
No	Increment	No Change

Table 1. Tuning decision table

avoid local maxima. We can obtain a measure of global network throughput (the objective function) in a manner similar to the way we obtain the global count of full virtual buffers (see Section 3). Nodes exchange the number of flits delivered in the past g cycles to measure throughput. If we consider the maximum possible delivered bandwidth of 1 flit/node/cycle, the count will not exceed $g * NodeCount$.

4.1 Hill Climbing

To automatically tune the threshold, we begin with an initial value based on network parameters (e.g., 1% of all buffers). We use intuition about the relationship between the number of full buffers and delivered bandwidth to specify a tuning decision table that indicates when the threshold value must be increased or decreased. Too low a value (A or lower in Figure 2) prevents us from reaching the peak by over throttling packet injection. In contrast, too high a value pushes us beyond the peak (C or higher in Figure 2), causing saturation just like a network without any congestion control.

Table 1 illustrates the tuning decision table. The two dimensions in the table correspond to observed network throughput and whether or not the network is currently throttled. We make a tuning decision once every *tuning period.* We say that there is a drop in bandwidth only if the the throughput in a tuning period drops to less than a specified fraction (we use 75%) of the throughput in the previous tuning period. We leave more complex schemes that adaptively modify the frequency of tuning according to network conditions as future work.

The tuning period is an exact multiple of the *gather-duration*. If the tuning period is very large, there is likely to be slow and inefficient tuning leading to network underutilization or network saturation. If it is too small, short-lived crests and troughs in throughput could alter the estimate. However, in our experiments, we found that, for a reasonable range of values (32 cycles to 192 cycles) the performance did not alter significantly. In our experiments, we use a 96 cycle tuning period.

The tuning process proceeds as follows. If we observe a decrease in throughput (upper row), then we always decrease the threshold value. The decreased throughput is a result of either network saturation or a decrease in offered load. If the cause is saturation, then we must reduce the threshold value to bring the network out of saturation. If the offered load has decreased, then it is still safe to reduce the threshold since the system is not over-throttling.

If the system is throttling and there is no drop in throughput, we optimistically increase the threshold. If there is no drop in throughput after the increase, our optimism was justified and the lower threshold value was over-throttling. If we exceed the saturation point because of the increase in threshold, the observed bandwidth decreases and we again reduce the threshold value. Finally, if throttling is not occurring and there is no decrease in throughput, we do not change the threshold value.

We use constant additive increments and decrements to update the threshold value. There could be other, more sophisticated, algorithms to tune the threshold that improves the tuning mechanism by reducing the time to reach the "sweet spot". We find constant additive tuning adequate for effective self-tuning and do not explore other, more complicated methods in this paper. For a reasonable range of values, (1% to 4% of all buffers) performance is insensitive (within 4%) to variations in increment/decrement values. There is marginally better performance when the decrement is higher than the increment. We use an increment of 1% of all buffers and a decrement of 4% of all buffers. For our 16-ary, 2-cube network this corresponds to an increment of 30 and a decrement of 122.

Since we're only comparing throughput observed in each *tuning period* to the throughput observed in the previous *tuning period*, it is possible that if the bandwidth drop due to saturation happens gradually, we will not treat it as sufficient to trigger a decrease in threshold value. Thus, the network "creeps" into saturation and hits a local maximum. The hill climbing technique, as currently proposed fails to move back from this local maximum. Increasing the threshold beyond this keeps the network in saturation with no further drop in bandwidth. In the next section, we describe a method to scale back the threshold away from the local maximum.

4.2 Avoiding Local Maxima

To avoid settling at local maxima, we remember the conditions that existed when maximum throughput was achieved. To do this, we keep track of the maximum throughput (max) achieved during any single tuning period and remember the corresponding number of full buffers (N_{max}) and threshold (T_{max}).

If the throughput in any tune-period drops significantly below the maximum throughput, our technique tries to recreate the conditions that existed when maximum throughput was achieved. We do this by setting the threshold to $min(T_{max}, N_{max})$. If N_{max} is higher than T_{max}, it means that the network was throttling with a threshold value of T_{max} when it achieved the maximum observed throughput. In this case, we set the threshold to T_{max} so that the network can throttle new packets and drain existing packets till it reaches the desired load level. If, on the other hand, N_{max} is smaller than T_{max}, then setting the threshold to N_{max} is a better choice because it is possible that T_{max} is not low enough to prevent saturation. This guarantees that we're not stuck at a local maximum after the network saturates.

It is possible that the threshold value which sustains high throughput for one communication pattern is not low enough to prevent saturation for another communication pattern. Our congestion control mechanism detects and adapts the threshold to such changes. If we find that we reset the threshold to $min(T_{max}, N_{max})$ for r consecutive tuning-periods, this means that even the $min(T_{max}, N_{max})$ value is too high to prevent saturation, and we must recompute max value. In this case, we reset max to zero and start the maximum locating all over again. This ensures that our threshold adapts to changing communication patterns. We use $r = 5$ in our experiments.

4.3 Summary

The above discussion provides a general overview of a technique we believe can provide a robust, self-tuned congestion control. Our scheme gathers full-buffer counts and throughput measurements every 32 cycles. The full-buffer counts are used to estimate current congestion using linear extrapolation. This estimate is compared to a threshold to decide whether we throttle new packets or not. We use a hill climbing algorithm to update our threshold every 96 cycles in increments and decrements of 1% and 4% of total buffer count, respectively. Our hill climbing algorithm, when used alone, is susceptible to settling on local maxima after network saturation. Our scheme includes a mechanism to prevent this from happening by remembering maximum (max) observed throughput. Finally, we recompute the maximum (max) if we reset the threshold for $r = 5$ consecutive tuning periods.

On a high level, our scheme is somewhat analogous to TCP's self-tuning congestion control. Both have an idea of what the network performance should be. Expected round-trip time (RTT) in the case of TCP and max throughput in our case. Both schemes allow offered load to incrementally increase as long as network performance is not penalized. The sliding window size increases as long as no packets are dropped in the case of TCP and threshold increases as long as there is no decrease in throughput in our case. Both techniques take corrective action if network performance suffers. TCP reduces its window size and our scheme either decrements the threshold or resets it to $min(N_{max}, T_{max})$. Finally, both schemes periodically refresh their estimate of network performance. TCP recomputes expected round-trip-time if packets are dropped, whereas our scheme recomputes max, N_{max} and T_{max} if max is stale, i.e. if there are r consecutive corrective actions.

5 Evaluation

This section uses simulation to evaluate our congestion control technique. We begin by describing our evaluation methodology. This is followed by our simulation results.

5.1 Methodology

To evaluate our congestion control scheme, we use the `flexsim` [27] simulator. We simulate a 16-ary, 2-cube (256 nodes) with full duplex physical links. Each node has one injection channel (through which packets sent by that node enter the network) and one delivery channel (through which packets sent to that node exit the network). We use three virtual channels per physical channel and edge-buffers (buffers associated with virtual channels) which can hold eight flits.

The router's arbiter is a central resource which only one packet can use at a time and there's a one cycle routing delay per packet header. Packets obtain control of the router's arbiter on a *demand-slotted round-robin distribution*. This is not a bottleneck because routing occurs only for the header flit of a 16-flit packet. The remaining flits simply stream behind the header flit along the same switch path. It takes one cycle per flit to traverse the cross-bar switch and one cycle per flit to traverse a physical link.

We evaluate our congestion control mechanism with both deadlock avoidance and deadlock recovery mechanisms. Deadlock avoidance uses the method proposed by Duato [8] with one escape virtual channel using oblivious dimension-order routing. We use the Disha [17] progressive deadlock recovery scheme with a time-out of 8 cycles.

All simulations execute for 60,000 cycles. However, we ignore the first 10,000 cycles to eliminate warm-up transients. Most results are presented in two parts: normalized delivered throughput (accepted flits/node/cycle) and average packet latency versus offered load in terms of packet injection rate.

The default load consists of each node generating 16 flit packets at the same fixed rate. We consider four different communication patterns, *uniform random, bit-reversal, perfect-shuffle and butterfly*. These communication patterns differ in the way a destination node is chosen for a given source node with bit co-ordinates $(a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$. The bit co-ordinates for the destination nodes are $(a_{n-2}, a_{n-3}, \ldots, a_0, a_{n-1})$ for *perfect shuffle*, $(a_0, a_{n-2}, \ldots, a_1, a_{n-1})$ for *butterfly* and $(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$ for *bit-reversal*. In this paper, we present results for steady loads with *uniform random* communication pattern. We also present results for a bursty load with the various communication patterns. Results for steady loads with *bit-reversal, perfect shuffle and butterfly* communication patterns are presented in greater detail in a technical report [28].

We use synthetic workload, instead of full-blown multiprocessor workloads, for three reasons. First, our simulation environment cannot handle full-blown multiprocessor workloads. Second, our packet generation frequency corresponds to realistic miss rates in databases and scientific applications, which gives us confidence in our results. Third, our synthetic workloads nicely demonstrate the problem of network saturation and avoids interactions with application-specific features.

For our network (with 3072 buffers), 12 bits are enough to count all buffers in the network. Our configuration needs 13 bits to represent the maximum possible aggregate throughput $g * NodeCount * MaxTraffic = 32*256*1 = 8192$ flits). Thus, we need a total of 25 bits for the sideband signals. However, in our technical report [28], we show that we can send these 25 bits using 9-bit sideband channels with very little performance degradation.

For comparison, we also simulate the At-Least-One (ALO) [2] congestion control scheme. ALO estimates global network congestion locally at each node. If at least one virtual channel is free on every *useful* [2] physical channel or if at least one *useful* physical channel has all its virtual channels free, then packet injection is allowed. Otherwise, new packets are throttled.

5.2 Simulation Results

The primary conclusions from our simulations are:

- Our technique provides high performance consistently across different communication patterns and offered load levels.
- Our technique outperforms an alternative congestion control technique that uses local estimates of congestion.
- Our self-tuning technique adapts the threshold dynamically to varying workloads and to bursty traffic.

The remainder of this section elaborates on each of these items.

5.2.1 Overall Performance

We begin by examining the performance of a complete implementation, as described in Sections 3 and 4. Figure 3 shows the bandwidth and latency for a *uniform-random* traffic pattern for both deadlock recovery (a & b) and deadlock avoidance (c & d). Note the logarithmic scale used on the y-axis for the latency graphs (b & d).

The curve for the base case illustrates the network saturation problem. As load increases, the network throughput increases to a certain extent. However, at saturation, there is a sudden drop in throughput since only the escape channels are available to drain deadlocks. The configuration with deadlock recovery has lower bandwidth at saturation because Disha deadlock recovery requires that a packet obtain exclusive access to the deadlock-free path. In contrast deadlock avoidance schemes can break multiple deadlock cycles concurrently.

The results in Figure 3 clearly show the key point that our congestion control technique ($Tune$) is stable at high loads. The ALO congestion control scheme improves performance in the early stages of congestion for the deadlock avoidance case, but it does exhibit severe performance degradation eventually. Our scheme, however, maintains latency and throughput close to the peak values.

Throughout this paper, we assume a side-band hop delay of 2 cycles. Our technical report contains simulation results that quantify the effect of varying this delay [28].

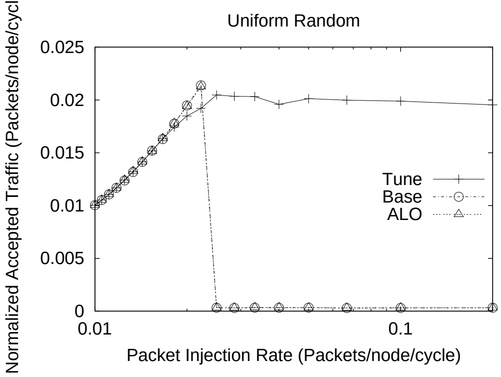
5.2.2 Self-Tuning

In this section, we demonstrate two important aspects about our self-tuning technique. First, we show the importance of having a congestion control mechanism that adapts to the congestion characteristics of different communication patterns. This is followed by an examination of the hill-climbing algorithm and the scheme to avoid local maxima.

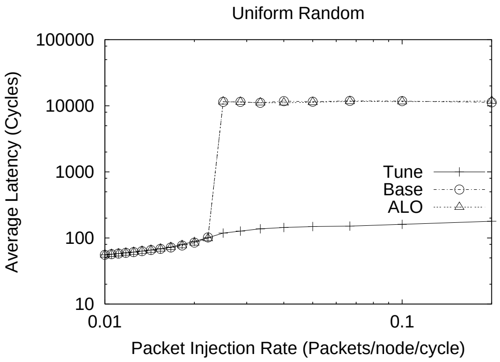
Recall from Figure 1 that saturation occurs at different levels of offered load for random and butterfly communication patterns. These different levels of offered load correspond to different buffer occupancies in the network. If saturation was occurring at the same buffer occupancies for different workloads, a well-chosen, single, static threshold could prevent network saturation. To show that this is not

[2] *useful* is an output channel that can be used without violating the minimal-routing constraint.

Deadlock Recovery

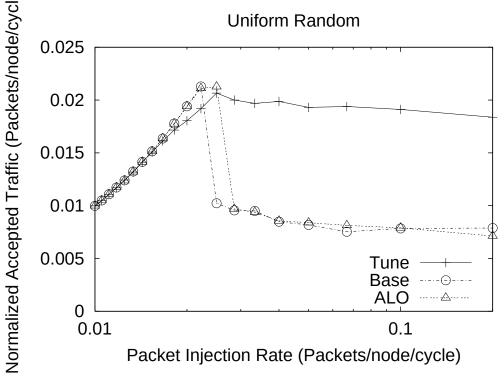


(a) Delivered Throughput vs. Offered Load

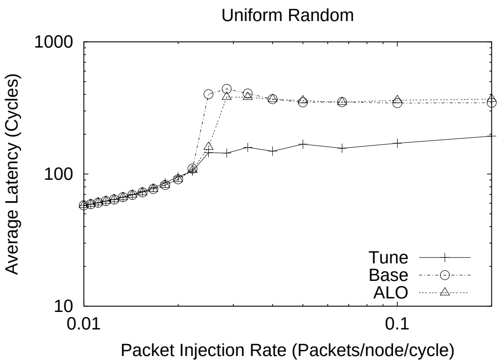


(b) Average Latency vs. Offered Load

Deadlock Avoidance



(c) Delivered Throughput vs. Offered Load



(d) Average Latency vs. Offered Load

Figure 3. Overall Performance With Random Traffic

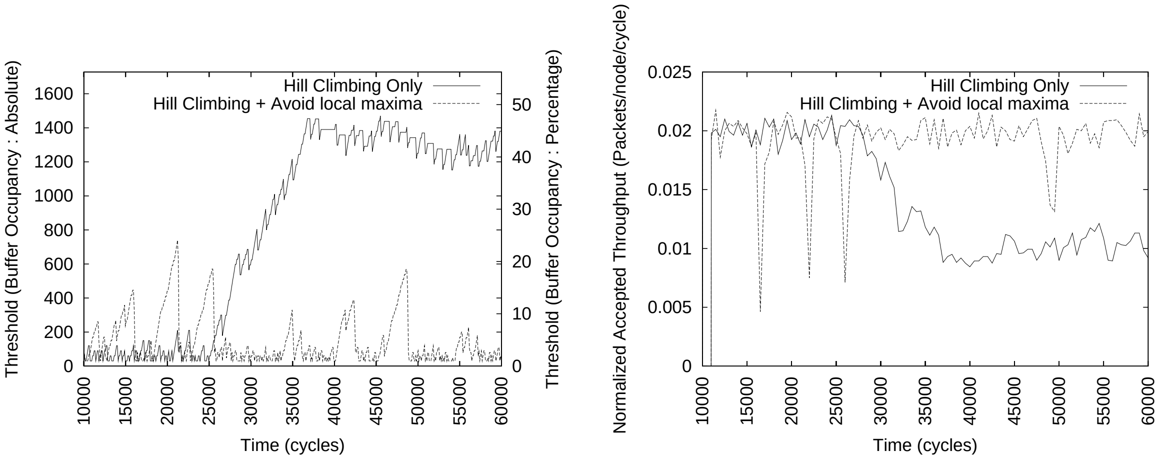
so, Figure 5 compares the performance on the deadlock recovery network configuration of a congestion control mechanism with static thresholds to our scheme.

We consider uniform random (the four solid lines) and butterfly (the four dashed lines) communication patterns. We see that a static threshold of 250 (8% buffer occupancy) works very well for random traffic but the same threshold is unable to prevent saturation for the butterfly communication pattern. In contrast, a static threshold of 50 (1.6% buffer occupancy) works well for the butterfly communication pattern but over-throttles the random traffic. This indicates that the buffer occupancy at which the network saturates is not uniform across communication patterns. Therefore, it is necessary to have a self-tuning mechanism that adapts the threshold as communication patterns change.

To understand the behavior of our self-tuning technique, we analyze its operation for a specific configuration. As stated in Section 4, we use a *gather-period* (g) of 32 cycles, a tuning period of 96 cycles, an increment of 1% of all virtual channel buffers and a decrement of 4% of all virtual channel buffers. The load is of uniform random distribution with a packet regeneration interval of 10 cycles and we use the deadlock avoidance configuration. With these parameters, Figure 4(a) shows the tuning of the threshold over time for the duration of the simulation. Recall, the first 10,000 cycles are ignored to eliminate start-up transients. Figure 4(b) shows the throughput achieved over the same interval.

The hill climbing mechanism tries to increase the threshold as long as there is no decrease in bandwidth and tries to scale back when bandwidth decreases. But it can settle down at a local maximum when the decrease in bandwidth happens gradually. When this occurs, the network "creeps" into saturation and throughput falls.

Without a mechanism to avoid local maxima, the hill climbing algorithm can settle on a local maximum corresponding to deep saturation. The solid line in Figure 4 shows this behavior. A gradual drop in throughput begins



(a) Threshold vs. Time

(b) Throughput vs. Time

Figure 4. Self-Tuning Operation : An Example

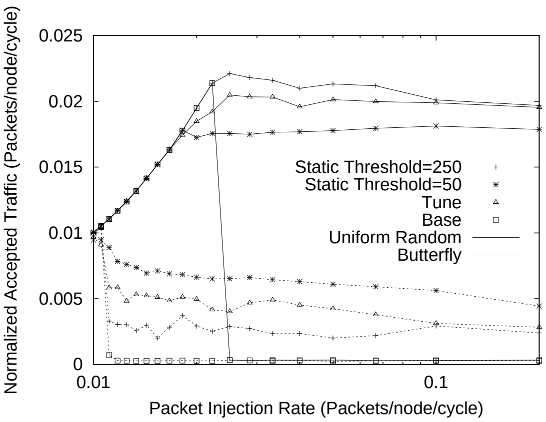


Figure 5. Static Threshold vs. Tuning.

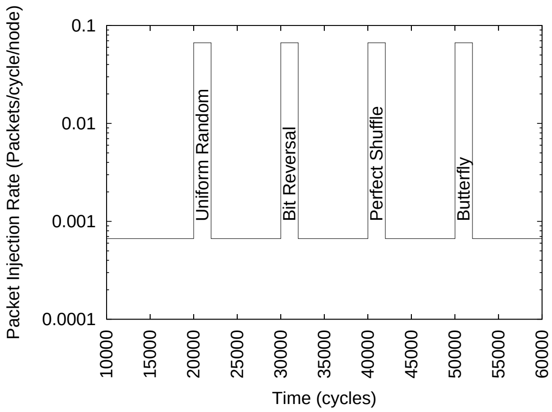


Figure 6. Offered bursty load

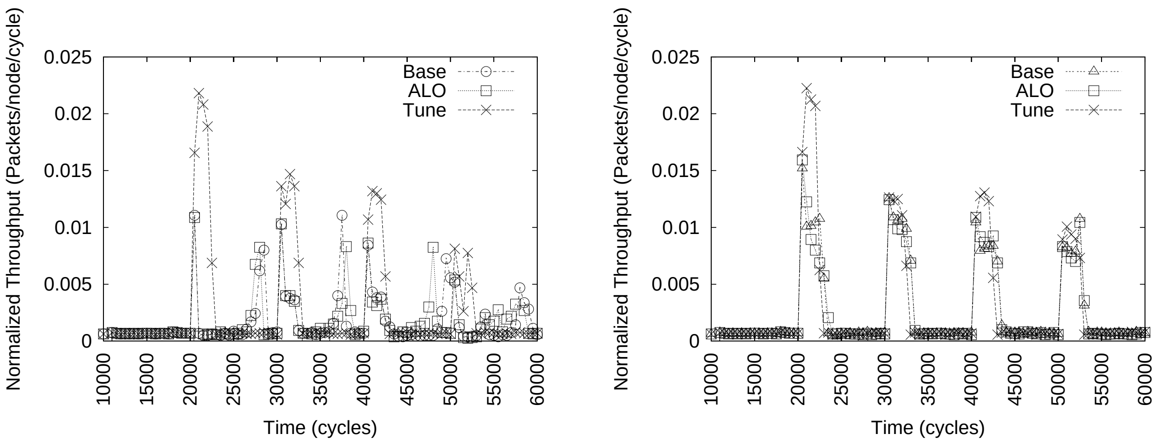
at approximately 26,000 cycles. Recall that we decrement the threshold only when there is a throughput drop of 25% or more in any tuning period. We see, in Figure 4(a), that although there are many decrements, the gradual nature of the decrease in throughput results in an overall rise in the threshold, eventually saturating the network.

The dashed line in Figure 4 shows the behavior of our scheme to avoid local maxima. The sharp dip in the threshold value (specifically the one at approximately 26,000 cycles) illustrates the corrective action taken when the network "creeps" towards saturation. As a result, we avoid saturation and sustain higher throughput.

5.2.3 Bursty Traffic

To confirm that our self-tuning mechanism works well under varying load, we use a bursty load created by alternating low loads and high loads. In addition, we also change the communication pattern for each high load burst. The offered bursty load is shown in Figure 6. In the low load phase, the communication pattern is *uniform random* and every node tries to generate one packet every 1,500 cycle period (corresponding to a packet injection rate of 0.00067 packets/node/cycle). In the high load phase, every node tries to generate a packet every 15 cycles (corresponding to a packet injection rate of 0.067 packets/node/cycle). The communication pattern in each of these high load bursts is different and is indicated in Figure 6.

Figure 7(a) and Figure 7(b) show the delivered throughput with bursty load for the deadlock recovery and the deadlock avoidance configurations, respectively. With deadlock recovery, the average packet latency for $Base$, ALO and $Tune$ configurations are 2838 cycles, 2571 cycles and 161 cycles respectively. With deadlock avoidance, the average packet latency for $Base$, ALO and $Tune$ configurations are 520 cycles, 509 cycles and 163 cycles respectively. In the high-load phase, our congestion control consistently delivers sustained throughput and predictable latencies. The



(a) w/ Deadlock Recovery (b) w/ Deadlock Avoidance

Figure 7. Performance with Bursty Load : Delivered Throughput

ALO scheme and the base scheme initially ramp up the throughput but throughput collapses soon thereafter due to network saturation.

The deadlock recovery results exhibit an interesting phenomenon in the $Base$ and ALO schemes (Figure 7a). There are small bursts in throughput long after the offered load is reduced. This is because the network absorbs the heavy offered load but goes into deep saturation with many deadlock cycles. We observe this happening approximately between 20,000 and 21,000 cycles in Figure 7(a). There is a period when network packets are draining through the limited escape bandwidth available (approximately between 21,000 and 27,000 cycles). It is only when the deadlock cycles break that full adaptive routing begins again. The network then drains quickly showing the spurt in throughput (approximately between 27,000 and 29,000 cycles).

6 Conclusion

Interconnection network saturation, and the commensurate decrease in performance, is a widely known problem in multiprocessor networks. Limiting packet injection when the network is near saturation is a form of congestion control that can prevent such severe performance degradation. Ideal congestion control implementations provide robust performance for all offered loads and do not require any manual tuning.

The primary contribution of this paper is the development of a robust, self-tuned congestion control technique for preventing network saturation. Two key components form the basis for our proposed design. First, we use global knowledge of buffer occupancy to estimate network congestion and control packet injection. When the number of full buffers exceeds a tunable *threshold*, packet injection is stopped. When congestion subsides, the full buffer count drops below the *threshold* and packet injection restarts.

The second piece of our solution is a self-tuning mechanism that observes delivered network throughput to automatically determine appropriate threshold values. Inappropriate thresholds can either over-throttle the network, unnecessarily limiting throughput, or under-throttle and not prevent saturation. A self-tuning mechanism is important since no single threshold value provides the best performance for all communication patterns.

Using simulation, we show that our design prevents network saturation by limiting packet injection. The results also show that our technique is superior to an alternative implementation that uses local estimates of congestion because global information can detect congestion in its early stages. We show that different communication patterns require different threshold values to prevent saturation without unnecessarily limiting performance, and that our self-tuning mechanism automatically adjusts to changes in communication patterns.

Acknowledgments

We would like to thank Amin Vahdat, Jeffrey Chase, Gershon Kedem, Kenneth Yocum, Joel Emer, Mark Hill and the anonymous referees for their feedback and suggestions to improve this paper. We would like to thank Timothy Pinkston, Yungho Choi, and Yong Ho Song for providing us with the `flexsim` network simulator and helping us debug our simulation code. We would also like to thank Jose Duato for giving us pointers to his work on network saturation.

This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637, CDA-95-12356, and EIA-99-72879, Career Award MIP-97-02547, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. The

views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] D. Basak and D.K. Panda. Alleviating Consumption Channel Bottleneck in Wormhole-Routed k-ary n-cube Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):481–496, May 1998.

[2] E. Baydal, P. Lopez, and J. Duato. A Simple and Efficient Mechanism to Prevent Saturation in Wormhole Networks. In *Proceedings. 14th International Parallel and Distributed Processing Symposium*, pages 617–622, 2000.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *Journal of Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[4] W. J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[5] W. J. Dally and C. L. Seitz. The TORUS routing chip. *Journal of Distributed Computing*, 1(3):187–196, October 1986.

[6] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[7] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), October 1999.

[8] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.

[9] J. S. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, October 1999.

[10] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communications Review*, 24(5):10–23, October 1994.

[11] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[12] P. T. Gaughan and S. Yalamanchili. Adaptive Routing Protocols for hypercube Interconnection Networks. *IEEE Computer*, pages 12–22, May 1993.

[13] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM '88 Symposium*, pages 314–329, August 1988.

[14] R. Jain. Congestion Control and Traffic Management in ATM networks: Recent Advances and a Survey. *Computer Networks and ISDN Systems*, October 1996.

[15] P. Kermani and L. Kleinrock. Virtual Cut-Through : A New Computer Communication Switching technique. *Computer Networks*, 3:267–286, 1979.

[16] J. H. Kim, Z. Liu, and A. A. Chien. Compressionless Routing: A Framework for Adaptive and Fault-Tolerant Routing. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[17] Anjan K.V. and T.M. Pinkston. An Efficient, Fully Adaptive Deadlock Recovery Scheme : Disha. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 201–210, June 1995.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[19] P. Lopez, J. M. Martinez, and J. Duato. DRIL : Dynamically Reduced Message Injection Limitation Mechanism for Wormhole Networks. In *International Conference on Parallel Processing*, pages 535–542, August 1998.

[20] P. Lopez, J. M. Martinez, J. Duato, and F. Petrini. On the Reduction of Deadlock Frequency by Limiting Message Injection in Wormhole Networks. In *Proceedings of Parallel Computer Routing and Communication Workshop*, June 1997.

[21] L.-S. Peh and W.J. Dally. Flit-Reservation Flow Control. In *Proceedings of the Sixth Internation Symposium on High Computer Architecture*, pages 73–84, January 2000.

[22] G. F. Pfister and V. A. Norton. Hot-Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[23] K.K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, 8(2):158–181, 1990.

[24] S. Scott and G. Sohi. The Use of Feedback in Multiprocessors and its Application to Tree Saturation Control. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):385–398, October 1990.

[25] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh Internation Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[26] A. Smai and L. Thorelli. Global Reactive Congestion Control in Multicomputer Networks. In *5th International Conference on High Performance Computing*, pages 179–186, 1998.

[27] The Superior Multiprocessor ARchiTecture (SMART) Interconnects Group, Electrical Engineering - Systems Department, University of Southern California. *FlexSim*. http://www.usc.edu/dept/ceng/pinkston/tools.html.

[28] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee. Self-Tuned Congestion Control for Multiprocessor Networks. Technical Report CS-2000-15, Duke University, November 2000.

[29] S. Warnakulasuriya and T.M. Pinkston. Characterization of Deadlocks in Interconnection Networks. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.