Avoiding Tree Saturation in the Face of Many Hotspots with Few Buffers

Bradley C. Kuszmaul
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
email: bradley@mit.edu

William H. Kuszmaul
MIT PRIMES and Stanford University
email: william.kuszmaul@gmail.com

***Abstract*—In a multistage network, hotspots induce tree saturation. The known solutions employ a variety of techniques, including combining (which works only for certain kinds of messages), feedback damping (which appears to provide low utilization in the absence of hot spots), and large numbers of buffers. In practice, the approach used today is to provide large numbers of buffers: in a P-processor system, the rule of thumb appears to be to provide $10P$ buffers, but $10P$ buffers may be too expensive for systems containing 10^5 or more processors. Even employing $\Omega(P)$ buffers does not appear to provide any guarantees, however. We show that by organizing the switches so that the messages addressed to a particular processor can use only certain of the buffers, many hotspots can be tolerated with few buffers. For example, a switch with $O(\log P)$ buffers can tolerate a single hotspot with probability 1, and allows the first few hotspots to have a large number of buffers before being declared a hotspot. A switch with B buffers can be organized so that it blocks a particular non-hotspot message with probability less than $O(1/s)$ if there are $O(B/\log s)$ hotspots, and can handle a factor of $O(B(\log\log s)/\log s)$ more hotspots before the probability becomes a constant. A similar approach can also be used to improve caching behavior in a multithreaded system in which one of the threads tries to consume all of the cache.**

I. Introduction

Large-scale computing systems typically employ multistage interconnection networks to interconnect their processors. These systems can suffer from ***hotspot contention***, however [1]. Hotspots arise when source processors collectively send too many messages to a particular destination processor—the destination processor falls behind trying to receive messages, and the messages back up into the network filling up buffers in the switches leading to the destination. Then the switches leading to those switches fill up, and eventually the congestion propagates backward through the network, forming a tree rooted at the hot destination of routing nodes with full buffers. This saturation pattern is sometimes called ***tree saturation*** [1]. Hotspots require very little nonuniform traffic and onset can be very fast and can take a long time to alleviate [2]. Data center and supercomputer switches urgently need effective congestion management to avoid performance problems [3].

Tree saturation becomes a bigger problem as a system grows to encompass more and more processors. Tree saturation originally was understood as a problem even on networks containing as few as 100 processors [1], but a series of techniques has mitigated tree saturation, at least on networks with as many as thousands of processors.

Several techniques have been proposed to solve hotspot contention. These techniques can be divided into four categories: ***combining***, ***feedback***, ***buffering***, and ***counting***.

The combining solution involves organizing the system so that switches find two messages that are destined for the same processor, and combine them into one message. Although the rules for combining messages have been worked out for certain cases (e.g., to implement shared memory [4]), many kinds of messages cannot combine. Even for combinable messages, one problem that shows up is that messages going to the same address might not ever meet each other, even if there is a hotspot. Ranade showed a theoretically effective way to implement shared-memory on a butterfly [5], but combining has yet to be implemented in any large-scale or commercially important systems.

The feedback approach aims to stop messages destined to a hotspot at their sources (e.g., [6], [7]). Feedback schemes notice that there is a hotspot, and provide information back to the sources to slow them down. This kind of negative-feedback control system seems to require difficult tuning to avoid oscillation, and even when successful, appears to underutilize the network [8].

The buffering approach involves adding more buffering to the internal nodes of the routing network. It has been long known that organizing the switches as a collection of FIFO's on input ports gives poor performance, due to head-of-line blocking. By using buffers, which allow an unblocked message to pass a blocked message, good performance can be achieved, however (e.g., see [9]). Systems that avoid head-of-line blocking require only a few buffers per input port to get good performance on random routing patterns.

Today's network architects appear to be doubling down on the buffering approach. For example, Bechtolsheim [3], [10] indicates that for a network containing P processors, a switch should contain enough buffering to hold $10P$ messages, but that the implied memory requirements seem too large to be practical.

One reason the use of many buffers does not necessarily solve the hotspot problem is that networks perform flow

This work was supported in part by NSF grants CCF-0937860, CCF-1162148, CNS-1017058, CCF-1314547.

control on the wrong units. For example, TCP performs flow-control on an individual ***flow***, which is the set of packets going in one direction as part of a single TCP connection. It takes a collusion of several flows to create a hotspot, however. On the other hand, the DCE standard [11] performs link-level flow control, and provides no way to stop a particular subset of messages: it's all or nothing across a link. This paper assumes that a fine-grained link-level flow control can be employed to stop particular messages.

The counting approach is exemplified by Dias and Kumar [12], who provide a guarantee against hotspots via careful buffer management. Their scheme enforces a rule that each input port may buffer only one message destined to a given processor. They simulated 4 buffers per input port with a 2×2 switch, and found in the face of a single hot spot, the network behaved reasonably well. Although they simulated only one hot spot, their approach can clearly handle more than one hot spot. Given B buffers per input port, a network can tolerate up to B hot spots before non-hotspot traffic is blocked. Our simulation results indicate that it can tolerate $O(B)$ hot spots with good performance, which is not surprising, since it is known that for random routing, only a few buffers are needed per input port.

Our approach can be thought of as a variant on the counting approach. We limit the number of buffers that hotspot messages can occupy, but with more flexibility to achieve better performance.

Given some buffering and flow control, we can provide two different guarantees under hotspot contention: a strong guarantee and a weak guarantee. To define these guarantees, we need to first define two classes of processors:

A hotspot is a processor that is receiving messages too slowly, causing messages to back up into the network. Usually a hotspot stems from too many messages being sent to a processor, but for our analysis we assign the blame to the recipient rather than the senders. We can model a hotspot processor as though it has stopped and fully saturated the tree of buffers that are available to the messages.

A coolspot is any processor that is not a hotspot.

Furthermore, we define

A complicit processor is one that has sent a message to a hot processor, and the message has not yet been delivered.

An innocent processor is any non-complicit processor. All the undelivered messages that it has sent are addressed to processors which are not hotspots.

Thus processors are either hot or cool and they are either complicit or innocent. All four cases are possible: for example a hot processor could be innocent if it is receiving messages too slowly but is not sending messages to a hot processor.

The ***strong guarantee*** is that any message addressed to a coolspot will be delivered efficiently. By "delivered efficiently", we mean that the message will be delivered about as quickly as if there were no hotspots. Since we need only a few buffers for random traffic, our goal is to keep a few buffers open for random traffic, even in the face of hotspot traffic. Since buffers used by coolspots are not problematic, in our analysis of results in this paper we will consider buffers to be full only if they are being used by a hotspot (and ignore buffer usage by coolspots).

The ***weak guarantee*** is that any message sent by an innocent processor to a coolspot will be delivered efficiently. On the other hand, a complicit processor may have its unrelated messages slowed down by the hotspot.

To begin, we introduce a way to implement a strong hotspot guarantee that can handle an arbitrary collection of hotspots, using P buffers in each switch.

The P-buffer strategy: Assign buffer i to processor i, and allow only messages addressed to processor i to be stored in buffer i. This approach requires some bookkeeping inside each switch, as well as fine-grained flow control so that we can stop the messages from one processor without stopping other messages. This approach also requires a buffer design that allows a switch to select an unblocked message to be forwarded to the next switch.

To get good performance, we may want to allocate $2P$ buffers or more, to facilitate the pipelining of messages and flow control. Given a link with a transmitter and a receiver, by the time the receiver sends flow control information back to the sender indicating that the messages for a processor should be stopped, the transmitter may have sent more messages for the same processor. Furthermore, we may want to be able to absorb several messages destined to a particular processor, since even under uniform random workloads, we often see several messages with the same destination in the network at the same time. For example, if every processor sends a message to a random processor, then we have a system that can be analyzed by P balls being thrown into P bins, and we know that some processor is likely to receive $\Theta(\log P / \log \log P)$ messages.

Using $\Omega(P)$ buffers per switch seems expensive and inefficient. It seems expensive, since with, say 10^6 processors and Ethernet jumbo frames of $10,240$ bytes, each switch requires 10GB of memory for P buffers per switch, or perhaps 100GB if we want $10P$ buffers per switch. It seems inefficient since most of those buffers will not be in use most of the time.

The approach we present in this paper is a variation of the P-buffer strategy: we restrict the messages destined for a particular processor to particular buffers, but we use fewer than P buffers. Define the number of buffers per switch as B. Assuming that hotspots persist for a while, then we expect that all the buffers that a processor can use will become filled up with messages. How many buffers should we allow each processor to use? We present systems that can provide one of two different answers to this question by following different approaches: The first approach is that each hotspot may consume up to a constant number, j, of buffers. In this case, we can provide a guarantee that the system can route if there are fewer than B/j hotspots.

In an alternative approach, we allow hotspots to consume more buffers when there are few hotspots than when there are many hotspots. We can provide a guarantee that if there are h hotspots, then there are expected to be Bc^h buffers available

for coolspot traffic for some number $c < 1$. We call switches that behave this way ***dampening switches***.

We describe two different mechanisms to achieve these goals: Counters and Randomized buffer assignment.

The counter mechanism works as follows. Each switch maintains a counter counting how many messages in its buffers are destined to each processor. The switch requires only space $O(B)$ to maintain these counters, since it needs not maintain counters for processors that do not have messages present. When the counter reaches its limit, the switch sends flow control information stopping messages from that processor from entering the switch.

The idea for randomized buffer assignment is as follows. The destination processor of a message is hashed to produce a set of buffers that the message is allowed to use. If there no such buffers are available, then that processor is blocked.

With either mechanism, if all the buffers fill up, the link-level flow control stops all new messages from arriving.

The rest of this paper is organized as follows. Section II describes how to implement a weak guarantee for a butterfly network with $O(\sqrt{P})$ buffers. Section III describes a simple counter system that allows for one to use fewer buffers than one would use in the P-buffer strategy, if one expects there to be under some number of hotspots. Section IV describes how to implement dampening switches, which are switches that give the first few hotspots more buffers, and later hotspots fewer buffers. Section V describes how to build a dampening switch using randomized buffer assignment that gets a strong guarantee against one hotspot using a hashing scheme and $O(\log P)$ buffers. Section VI describes how to get a strong guarantee against many hotspots using randomized buffer assignment. Section VII shows that most hash functions can tolerate quite a few hotspots (under certain conditions). Section VIII describes the flow control for randomized buffer assignment. Section IX presents a few simulation results that show that dampening switches can provide substantial benefits over simple counting switches. Section X concludes with a discussion of how these ideas can be applied to caches.

II. Butterfly Networks

To get warmed up, let's examine how to use only $O(\sqrt{P})$ buffers to provide an arbitrary *weak* hotspot guarantee for an obliviously-routed butterfly. Recall that the weak guarantee states that if any set of processors has stopped, then processors who aren't sending messages to any of those processors can continue. This arbitrary weak guarantee can be useful in a space-shared computing environment, in which disjoint sets of processors work on different tasks: we don't want one task to slow down another. The Connection Machine CM-5 [13] was an example of a commercial machine that provided this kind of guarantee as long as the different tasks were given disjoint subtrees in the fat tree network. In contrast, this result allows an arbitrary assignment of processors (instead of subtrees).

To illustrate how it works, we focus on a binary butterfly. A binary butterfly comprises $P \lg P$ two-input two-output switches. The switches are numbered with pairs (i, j) where $0 \le i < P$ and $0 < j < \lg P$, where switch (i, j) has outputs connected to the inputs of switch $(i, j+1)$ and $(i \oplus 2^j, j+1)$.

The key observation is that for messages traveling in the second half of the network, there are at most $\sqrt{P}$ different destinations to which a message can get. So we allocate $\sqrt{P}$ buffers in each node, and assign at least one buffer exclusively to each possible destination.

In the first half of the network, there are at most $\sqrt{P}$ different sources that could have gotten to that node, so we assign buffers according to the source.

Now if any innocent processor sends a message to a coolspot, the message will be able to make forward progress: In the first half of the network, the message will make forward progress because all of the source processor's previous messages can make progress. In the second half of the network, the message will make forward progress because all of the destination's messages will make progress.

This scheme can be extended to many other oblivious routers, but the number of buffers used cannot be reduced by much. Borodin and Hopcroft show that any oblivious router for a P node network with degree d must have worst-case routing time of $\Omega(\sqrt{P/d^3})$ [14]. This also provides a lower bound on the number of buffers needed to provide an arbitrary weak hot-spot guarantee. On the other hand, oblivious routers seem to be seldom employed, possibly because the same result states that oblivious routing can be slow. For the rest of this paper we consider solutions that can work even on adaptive routers in which there may be switches that can receive a message from any processor and forward it to any processor.

III. The Counter Method

Suppose that we want each destination (or conversely, each arrival) processor to have up to j messages present before we declare the processor as a hotspot (for some constant j). When we did this in the previous section in the context of butterfly networks, we j assigned specific buffers to each destination processor. If there are D destination processors, this requires jD buffers. But suppose we are confident that there will never be more than k hotspots, or that we are willing to block all traffic through a switch if there are k hotspots present at the switch. In this case, we can use the following protocol and get away with jk buffers instead.

For each switch, we maintain a hash table tracking how many messages are at the switch that are destined for a given processor. In order to save space, we only keep track of the processors towards which at least one message at the switch is destined. (So the hash table is of size at most $O(jk)$.) When a message arrives or departs, we increment or decrement the appropriate element of the hash table.

When all jk buffers are full, we declare an ***interdiction*** on all senders connected to the switch; they are no longer allowed to send any messages.

When the jk buffers are not all full, we maintain that the processors declared as hotspots are exactly the ones that have j messages present at the switch. Observe that flow control requires at most one communication between switches

per arrival or departure. Also, observe that if a message can be forwarded directly to an output port upon routing (with cut-through routing), then no bookkeeping is needed for the message.

IV. Dampening Switches

Suppose we design switches such that when there are k hotspots at a switch, we expect $B((B-j)/B)^k$ buffers not to be claimed by any of those hotspots. Such a switch design is called a ***dampening switch***.

The idea of a dampening switch is that as hotspots appear, each one is allowed to use fewer and fewer buffers. The first hotspots get to use a lot of buffers, but the switch can also handle a lot of hotspots. Since multiple hotspots can appear at once, a dampening switch cannot actually assign a given amount of hotspots to the nth hotspot that appears. Hence the definition of a dampening switch provided above.

It is worth noting why one might want to give hotspots many buffers when one has the resources to afford doing so. One example of when doing so is helpful is when switches near a processor go through brief periods of heavy traffic due for that processor. Suppose these switches accept, for example, $B/2$ of those messages as outstanding messages instead of declaring the processor as a hotspot after, for example, 3 of those messages. Then those messages may start clearing up before the traffic jam has a chance to cascade up the network to other switches farther away from the processor. As a consequence, one might want to make switches that are near a destination processor be dampening switches.

We introduce two ways to build a dampening switch, the ***modified counter method*** and the ***randomized method***. The Modified Counter Method operates as follows. At each switch, we keep a sorted array of how many buffers are being used by each of the processors that have at least one outstanding message at the switch. (Each element in the array consists of a processor number and the value by which the list is sorted. This value is referred to as the element's ***value***) We call this list the ***processor impact array***, PIA. Then each time we add or subtract an outstanding message to our switch, we do the following. We update the processor impact array. If every buffer is full, we declare every processor as a hotspot. Otherwise, we find the largest k such that the first k elements in the processor array list sum to at least $B-B((B-j)/B)^k$. (Note that k may be zero.) We call this k ***the hotspot counter***. We then make sure that the processors that we have declared as hotspots are the ones corresponding to those first k elements.

The Randomized Method operates as follows. For each processor, we hash it to a random j-tuple of integers from 1 to B. We then allow each processor to use only the buffers named in the j-tuple. It all such buffers are full, we declare the processor to be a hotspot.

The modified counter method requires $(B/j)\ln B$ time for flow control (discussed later in this section) while the randomized method requires j time (discussed in Section VIII). Thus if $j^2 < B \ln B$, then one will likely prefer the randomized method. The modified counter method can also be implemented to run in P time, which may be useful if for some reason P is very small (for example, in the context of caching which we discuss in Section X).

In Section VIII, we describe a flow control scheme for a slightly modified version of the randomized method. One of the strengths of the scheme is that upon receiving or sending an outstanding message, it only ever declares or undeclares one processor as a hotspot. On the other hand, the modified counter method appears to require one to be constantly declaring and undeclaring processors as hotspots. Indeed, suppose that a outstanding message destined for a hotspot processor is sent. Then that hotspot, along with all of the other hotspots that are using fewer buffers (of which it turns out there may be order $(B/j)\ln B$; this is the maximum number of hotspots before all buffers are full), are immediately undeclared as hotspots. Then, when a message comes in for one of them, they may all immediately be redeclared as hotspots. One would sort of expect that fairly often, the sending of a single message that is destined for a processor that has been declared a hotspot would change the number of hotspots by a factor of two for the modified counter method. On the other hand, even without clever flow control, one would expect that the number of hotspots under the randomized method to change only by j/B times the number of hotspots.

The following result is relevant for dampening switches.

Theorem 1: When a dampening switch has $(B/j)\ln B$ hotspots, one expects it to have only one of its buffers not in use.[1]

Proof. It suffices to show that $\log_{B/(B-j)} B \approx (B/j)\ln B$.

$$\log_{B/(B-j)} B = \ln B / \ln(B/(B-j))$$
$$= \ln B / \ln \frac{\frac{B}{j}}{\frac{B}{j}-1} \approx (B/j)\ln B.$$

Now we will briefly discuss flow control for the modified counter method (because of how it ends up relating to Theorem 1). First, observe that the processor impact array requires only $O(1)$ time to keep up to date each time a message arrives or departs. One keeps two additional data structures in memory: a hash table tracking the positions in the processor impact array of each processor that has messages at the switch; and an array I which in index i keeps track of the element of highest valued position that has value i in the processor impact array. Note that we only ever change values of elements of the processor impact array by 1. For example, when we increment the value of PIA$[t]$, where a is the old value, we swap PIA$[t]$ with PIA$[I[a]]$, decrement $I[a]$, give $I[a+1]$ a value if it previously had none (because no element of PIA had value $a+1$), and update the hash table.

The remaining computational work takes $(B/j)\ln B$ time (or P time if $P < (B/j)\ln B$). This is because we do not need to compute the hotspot counter except for when it will be valued at no more than $(B/j)\ln B$. Indeed, by Theorem 1, when it is greater than $(B/j)\ln B$, all buffers at the switch are full; in this case, we've told all of our neighbors to stop

[1] Note that [15] provides a formula which can be used to reach Theorem 1.

sending any messages to us, and we don't need to explicitly compute the hotspot counter. Observing this, one can easily calculate the hotspot counter (when necessary) in $(B/j)\ln B$ time (or P time when $P < (B/j)\ln B$), and then declare or undeclare hotspots as needed in $(B/j)\ln B$ time.

The following sections will be devoted to studying the randomized method.

V. One Hotspot, $O(\log P)$ buffers

The next two sections show how to solve the tree saturation problem for arbitrary networks using randomized buffer assignment. This approach could be used, for example, on Fat trees [16], which are now used widely in data center networks. This section shows how to provide a strong hotspot guarantee against a single hotspot with $O(\log P)$ buffers per processor.

The idea is illustrated by this small example. Consider a machine containing 6 processors numbered 0 through 5 with switches containing 4 buffers each, named A, B, C, and D. We allow each processor to use only two switches, as shown in this table:

Processor	Allowed Buffers			
0	A	B		
1	A		C	
2	A			D
3		B	C	
4		B		D
5			C	D

Since $\binom{4}{2} = 6$ we can arrange that each processor has a distinct set of buffers attached to it. Now if any single processor stops receiving, all other processors still have at least one buffer they can use, so their messages continue to make progress.

We denote by S_i the set of buffers assigned to processor i. Observe that if, at a switch, $S_i \neq S_j$ then even if processor j stops, then processor i will be able to continue to send messages through the switch. We could make all the switches use the same mapping (in which case, knowing that a processor can get through one switch means that it can get through all the switches), or different switches could use different mappings.

We can apply this idea to a system with P processors by using just enough buffers B so that $\binom{B}{B/2} \geq P$. In this case, each processor can be assigned a unique subset of $B/2$ buffers. It turns out that B is a little bit larger than $\lg P$. One bound for the central binomial coefficient is

$$\binom{2n}{n} = \frac{4^n}{\sqrt{\pi n}}\left(1 - \frac{c_n}{n}\right),$$

where $1/9 < c_n < 1/8$ for all $n \geq 1$. This implies

$$\binom{\lg P}{0.5\lg P} \approx \frac{P}{\sqrt{(\pi/2)\lg P}},$$

and we need just about $\lg\lg P$ more buffers. If we use $1 + \lceil \lg P + \lg\lg P \rceil$ buffers we get

$$\begin{aligned}\binom{\lceil 1+\lg P+\lg\lg P\rceil}{\lceil(1+\lg P+\lg\lg P)/2\rceil} &\approx \frac{4^{(1+\lg P+\lg\lg P)/2}}{\sqrt{\pi(1+\lg P+\lg\lg P)/2}}\\ &= \frac{2^{1+\lg P+\lg\lg P}}{\sqrt{\pi(1+\lg P+\lg\lg P)/2}}\\ &= \frac{2P\lg P}{\sqrt{\pi(1+\lg P+\lg\lg P)/2}}.\end{aligned}$$

We want

$$\binom{\lceil 1+\lg P+\lg\lg P\rceil}{\lceil(1+\lg P+\lg\lg P)/2\rceil} \stackrel{?}{\geq} P.$$

Since

$$2\lg P/\sqrt{\pi(1+\lg P+\lg\lg P)/2} \in o(1),$$

the approximation is swamped for large P. For small P, we can calculate, and it turns out that there are enough combinations of buffers to give every processor a distinct set.

One way to statically assign buffers to processors numbered from 0 to $P-1$ is as follows. We use a twice as many buffers to make the calculation easier: using $B = 2\lceil \lg P \rceil$ buffers, we use the following buffer assignment. Given a processor number from 0 to $P-1$, we look at its binary representation b. If the ith digit of b is zero, then we assign buffer number $2i$ to the processor, and otherwise we assign the buffer number $2i+1$. Each processor gets a unique set of buffers, so we are guaranteed to be able to handle any single hotspot.

That approach provides P unique combinations of distinct buffers, and that works fine if we want to precompute the unique subset of buffers for each processor to use. One can imagine that the routing switch does not know how many processors there are or what their identities are in advance, however. For example, the switch might discover ethernet MAC addresses dynamically. Even in this situation, the switch could maintain a table of the MAC's that it has seen, and use the precomputed buffers. This kind of system would require a table big enough to hold P MACs, and since a MAC is only 6 bytes, storing P MACs could be perfectly reasonable.

For some systems, it might make sense to avoid a processor table, however. Such a system could use randomly chosen subsets of buffers, employing a hash of the MAC address. Borrowing an idea from Bloom filters, we could compute two hashes a and b, and then choose buffer numbers the form $a+ib$. Since we are choosing subsets of buffers randomly, we need more than P distinct subsets. The birthday paradox tells us that roughly P^2 distinct sets is enough.

By using $3\lg P$ buffers per processor in each switch we can get more than P^2 different sets. The number of of such possible sets is

$$\binom{3\lg P}{1.5\lg P} \approx \frac{4^{1.5\lg P}}{\sqrt{\pi 1.5\lg P}} = \frac{P^3}{\sqrt{\pi 1.5\lg P}}.$$

With nearly $O(P^3)$ distinct combinations of buffers, the chance that two processors would pick the same buffer set is vanishingly small. And by adding a few more buffers, the odds can be made arbitrarily small.

Theorem 2: $O(\log(P/\epsilon))$ buffers yields less than ϵ probability of collision.

Proof sketch: The number of combinations is exponential in the number of buffers, and the probability of collision is less than P^2/C where C is the number of combinations.

Thus, even switches with $o(P)$ storage can provide a guarantee that a single hotspot will not stop messages from being delivered to any other processor, with high probability.

VI. Many Hotspots with High Probability

Section V showed how to tolerate a single hotspot with a logarithmic number of buffers. This section shows that we can tolerate many hotspots if we have a few more buffers. For example, we can tolerate $\Theta(B/\log s)$ hotspots with any particular coolspot processor being blocked with probability $O(1/s)$. Or, assigning j buffers to each processor, we can tolerate $(B/j)(\ln(j)-t)$ hotspots with any particular coolspot processor being blocked with probability $1/e^{e^t}$.

Suppose we have P processors and B total buffers. Let j represent the number of buffers assigned to each processor. In assigning buffers to processors, suppose that we simply pick a j-tuple of random integers for each assignment (possibily picking the same buffer more than once). Let k be the number of hotspots.

Let $g(B,j,k,P)$ be the expected number of buffers that are not assigned to any hotspot. Let $\beta(B,j,k,P)$ be the probability that a given coolspot processor is blocked by a given k hotspots. Let $f(B,j,k,P)$ be the probability that there are no coolspots blocked.

Lemma 3:
$$\beta(B,j,k,P) \approx (1-e^{-jk/B})^j.$$

Proof. The probability that a given buffer is not assigned to any hotspot is
$$\left(1-\frac{1}{B}\right)^{jk} \approx e^{-jk/B}.$$
Thus $g(B,j,k,P) \approx Be^{-jk/B}$.

Observe that $\beta(B,j,k,P) = (B-g(B,j,k,P))^j/B^j$. Thus $\beta(B,j,k,P) \approx (1-e^{-jk/B})^j$.

Lemma 4: Let $\epsilon > 0$. For B sufficiently large,
$$\beta(B,j,k,P) \leq (1-(e+\epsilon)^{-jk/B})^j.$$

Proof. The probability that a given buffer is not assigned to any hotspot is $\left(1-\frac{1}{B}\right)^{jk}$.

For B sufficiently large, $\left(1-\frac{1}{B}\right)^B \geq (e+\epsilon)^{-1}$. Therefore, for B sufficiently large,
$$\left(1-\frac{1}{B}\right)^{jk} = \left(1-\frac{1}{B}\right)^{B(jk/B)} \geq (e+\epsilon)^{-jk/B}.$$

Consequently, for such B, $g(B,j,k,P) > B(e+\epsilon)^{-jk/B}$.

Observe that $\beta(B,j,k,P) = (B-g(B,j,k,P))^j/B^j$. Thus $\beta(B,j,k,P) < (1-(e+\epsilon)^{-jk/B})^j$.

Recall that using the counter method, one can handle B/j hotspots before all processors, hotspots or not, are blocked by flow control. What if we have B/j hotspots using the hashing method? By Lemma 4, we have $\beta(B,j,B/j,P) \approx (1-e^{-1})^j \approx .632^j \approx 1/2^{.66j}$.

Another natural question is, for what k does $\beta(B,j,k,P)$ start to get large, and how quickly does it do so? To answer this question, we prove the following result, which is the most important result in this section.

Theorem 5: Let $\epsilon > 0$. Let $k = (B/j)(\log_{e+\epsilon}(j)-t)$ where $t < \log_{e+\epsilon} j$ and B is sufficiently large. Then $\beta(B,j,k,P) < 1/e^{(e+\epsilon)^t} < 1/e^{e^t}$.

Proof. By Lemma 4, $\beta(B,j,k,P) < (1-(e+\epsilon)^{-jk/B})^j = (1-(e+\epsilon)^{-\log_{e+\epsilon}(j)+t})^j$. It follows that $\beta(B,j,k,P) < 1/e^{j(e+\epsilon)^{-\log_{e+\epsilon} j+t}} = 1/e^{(e+\epsilon)^t}$.

This high-probability result implies that we can tolerate many hotspots with even with many buffers per assignment. For example, if $j = \ln^2 P$ and $t = \ln\ln P$ (note that $\ln j = 2\ln\ln P = 2t$), then the Theorem 5 yields $\beta(j,k,P,B) = 1/p$ when $k = B/(2j)\ln j$. In fact, if $j \in \Omega(\log^2 P)$ we can tolerate $O((B/j)\log j)$ hotspots. Also, note that regardless of our choice for j, we can tolerate $k = (B/j)\ln j$ hotspots, getting $\beta(j,k,P,B) = 1/e$. Although $1/e$ is only an expectation, not a high-probability bound, it says that the hash method is still pretty functional even when we give it a factor of $\ln j$ more hotspots than we can give the counter method.

There is one additional interesting case to consider, that where we pick j as the value that for a fixed P, B, k minimizes $\beta(B,j,k,P) \approx (1-e^{-jk/B})^j$. Taking the derivative, we want
$$\frac{\partial(1-e^{-jk/B})^j}{\partial j} = 0,$$
which implies
$$jk + (-1+e^{(jk)/B})B\ln(1-e^{-(jk)/B}) = 0,$$
which in turn implies
$$j = (B/k)\ln 2.$$
Thus we have $\beta(B,j,k,P) \approx (1-e^{-jk/B})^j = 1/2^j$.[2]

Given this, it is interesting to note the following results. In particular, Theorem 9 will be useful when proving the existence and frequency of optimal hash functions in the Section VII.

Lemma 6: Let $j = (B/k)\ln 2$. Then $\beta(B,j,k,P) \approx 1/2^j$.

Proof. By Lemma 4, $\beta(B,j,k,P) \approx (1-e^{-jk/B})^j = (1-1/2)^j = 1/2^j$.

Here's the second most important result in this section: It provides a high-probability bound for how many hotspots we can tolerate with a given probability.

Theorem 7: Given B buffers, one can pick j such that one can handle $k = O(B/\log s)$ hotspots with $\beta(B,j,k,P) \approx 1/s$.

Proof. Let $j = \log s$. By Lemma 6, we can handle $k = (B/j)\ln 2 \in O(B/\log s)$ hotspots with $\beta(B,j,k,P) \approx 1/s$.

Lemma 8: Let $j = (B/k)\ln 2$. Then $f(B,j,k,P) \approx 1/e^{(P-B\ln 2)/2^j}$.

Proof. Since $f(B,j,k,P) = (1-\beta(B,j,k,P))^{P-jk}$, we have $f(B,j,k,P) \approx \left(1-1/2^j\right)^{P-B\ln 2}$. In turn, $f(B,j,k,P) \approx 1/e^{(P-B\ln 2)/2^j}$.

Theorem 9: For a given value of s, let $k = B\ln^2 2/(\ln(P-B\ln 2)+\ln s)$ and $j = (B/k)\ln 2$. Then, $f(B,j,k,P) \approx (1-1/s)$.

[2]Note that [17] shows, in the context of Bloom filters, that if you know that you are inserting k items into a Bloom filter of size B, then the optimal number of bits to set is $j = (B/k)\ln 2$, achieving false-positive probability $1/2^j$.

Proof. Pick a d. If we want the largest k such that $f(B,j,k,P)$ gets as small as d (when we pick $j=(B/k)\ln 2$) then, by Lemma 8, we want

$$1/d = e^{(P-B\ln 2)/2^{(B/k)\ln 2}},$$
$$2^{(B/k)\ln 2}\ln(1/d) = P - B\ln 2, \text{ and}$$
$$(B/k)\ln 2 + \lg\ln(1/d) = \lg(P-B\ln 2).$$

If $d=(1-1/s)$, then

$$(B/k)\ln 2 - \lg s = \lg(P-B\ln 2), \text{ and}$$
$$k = \frac{B\ln 2}{\lg(P-B\ln 2)+\lg s} = \frac{B\ln^2 2}{\ln(P-B\ln 2)+\ln s}.$$

VII. Picking an optimal hash setup

In Section VI we presented high-probability results stating that a random hash function keeps coolspot processors from being blocked, even when many hotspots exist. However, it would be nice to have a 100% guarantee for some number of hotspots. We say a hash function is ***k*-perfect** if, assigning buffers with that hash function, no choice of k hotspots blocks any coolspot processor. In this section, we provide a result on the frequency of k-perfect hash functions and a construction for a $(j-1)$-perfect hash function (under certain constraints).

Theorem 10: Suppose $k(k+t+1) = B\ln^2 2/\ln P$. Then, picking $j=(B/k)\ln 2$, out of all the hash functions at most 1 out of P^t hash functions are not k-perfect.

Proof. Recall that the hash functions map processors to j-tuples containing not necessarily distinct numbers.

By Theorem 9, given a choice of a hash function and a choice of k hotspots, there is approximately a $1/s$ chance that the hotspots block any coolspot processor when k satisfies $k = B\ln^2 2/(\ln(P-B\ln 2)+\ln s)$. Observe that given a choice of a hash function, there are no more than P^k ways to choose k hotspots. Therefore, if $s=P^{k+t}$, then only $1/P^t$ of the choices for hash functions can be not k-perfect. In this case, $k = B\ln^2 2/(\ln(P-jk)+\ln P^{s+t})$. If we pick a slightly smaller $k = B\ln^2 2/(\ln P + \ln P^{s+t})$, then this yields $k(k+t+1) = B\ln^2 2/\ln P$.

A simple and important implication of Theorem 10 is that we can pick j and k both in $\Theta(\sqrt{B/\ln P})$, with fewer than 1 in P hashes not being k-perfect.

In the remainder of the section, we construct a $(j-1)$-perfect hash function, the main constraint for which is that $P \le B^2/(j^2\ln(B/j))$. Observe that given P and B (meeting certain easy constraints), we can thus construct a $(j-1)$-perfect hash function for $j = B/\sqrt{P\ln B}$. In order to construct such a hash function, we design its range such that any two distinct j-tuples in the range overlap in at most one value, and such that each value in a given j-tuple is distinct.

Let $j,B\in\mathbb{N}$ such that $j|B$. Let $Q_{j,B}$ be a set of integers such that

1) $\text{LCM}(q,q') > qj$ for all $q,q'\in Q_{j,B}$;
2) $q \le B/j$ and $\text{GCD}(q,B)=1$ for all $q\in Q_{j,B}$.

Let

$$H_{j,B} = \{\{rqj+tq \bmod B | t\in[1..j]\} | q\in Q_{j,B}, r\in[0,B/j]\}.$$

Observe that $|H_{j,B}| = |Q_{j,B}|B/j$.

Given j and B, we can pick $H_{j,B}$ as the range of a hash function with domain $[1,|H_{j,B}|]$.

Theorem 11: If $a,b\in H_{j,B}$ are distinct then $|a\cap b|\le 1$.

Proof. We have

$$a = \{rq(j-1)+tq | r\in[0,B/j]\},$$
$$b = \{rq'(j-1)+tq' | r\in[0,B/j]\},$$

where $q,q'\in Q_{j,B}$. If $q=q'$, then since $\text{GCD}(q,B)=1$ and $j|B$, $|a\cap b|=0$. Suppose $q\ne q'$. Then since $\text{LCM}(q,q')$ and B are greater than both qj and $q'j$, $|a\cap b|\le 1$.

The next natural question to ask is how large we can construct $Q_{j,B}$ to be. For simplicity, we pick B to be a power of two. One simple construction is to pick $Q_{j,B}$ to contain

- for each odd prime q satisfying $\sqrt{j} < q \le B/j$, q;
- for each odd prime $q \le \sqrt{j}$, the smallest power of q greater than $\sqrt{j}$.

Since any two elements of the proposed $Q_{j,b}$ are relatively prime, each greater than $\sqrt{j}$, and each relatively prime to B, it follows that this set of elements does indeed satisfy the requirements of $Q_{j,B}$. Note that $|Q_{j,b}| = \pi(B/j)-1 \approx B/(j\ln(B/j))$. (Recall that $\pi(x)$ is the number of primes less than or equal to x, and $\pi(x)\approx x/\ln x$.) Thus $|H_{j,b}| = |Q_{j,B}|B/j \approx B^2/(j^2\ln(B/j))$. So, we have a hash function allowing at most one overlap between the buffers assigned to any two distinct processors; and the main requirement for this hash function is that $P\le B^2/(j^2\ln(B/j))$.

VIII. Flow control

The previous sections showed that switches can perform local management of messages using hashing to assign messages to buffers. When a switch determines that a processor can no longer be accommodated in its buffers, it must stop further messages from arriving. This section explains how to perform the bookkeeping and interswitch communication for flow control.

The flow control and buffer management should satisfy these requirements:

1) Perform $O(j)$ work per message, when each processor hashes to j buffers. (Any flow-control mechanism will take, in the worst case, $\Omega(j)$ probes into a table since it must look into the j possible buffer locations to find a free location.)
2) Don't incur latency: (that is, if there is little or no contention, then the switch must not delay messages.)
3) Send flow-control information across a communication link only when the arrival (or departure) of a message makes a processor into a hotspot (or coolspot).
4) Provide a single buffer pool, rather than one for every input port.

A. Sender calculates

We first present a simple scheme which satisfies only the first two requirements above. This scheme is simple to implement and easy to understand. In this scheme, we will send a little more flow-control information and we have a buffer

pool for every input port. Consider a communication link connecting a sender switch to a receiver switch. In this case, the sender will do all the calculations of buffer management for the receiver.

What the receiver does: The receiver receives a message, buffers it, and later forwards it (or possibly forwards it immediately without buffering, e.g., to effect cut-through routing [18]). Whenever the receiver forwards a message it informs the sender of the processor number of the forwarded message.

What the sender does: Keep a bitmap tracking which buffers are allocated to each message held by the receiver, and also keep a table tracking for each processor, which buffers are being used by messages destined to the processor.

It turns out that by having the receiver perform the bookkeeping, and using a slightly more complex scheme, we can reduce the flow control information and use a single buffer pool for the entire switch, reducing the number of buffers required by a factor of $\Omega(\Delta)$, the degree of the switch.

B. Receiver calculates

Here we present a second scheme which satisfies all four requirements. In this scheme, we have only a single set of buffers shared among all input ports. We have an extra set of B buffers called ***emergency-backup buffers***. Hence this system employs a total of $2B$ buffers instead of B buffers.

At any given point the receiver has declared a subset of the processors to be ***embargoed***. An embargoed processor is one whose assigned buffers are all full (possibly of unrelated messages) and is using an emergency-backup buffer. The sender does not transmit messages destined to embargoed processors. The list of embargoed processors is known by the receiver, and the sender must be kept up to date every time the list changes. In our implementation, the list size changes by at most one element whenever a message arrives or departs.

When all the emergency-backup buffers fill, the receiver declares an ***interdiction*** on all the senders connected to it, which stops all messages from being sent across the respective communication links.

When a message arrives destined to a processor, we put the message into one of the buffers assigned to the processor. If all those buffers are full, then we place the message into an emergency-backup buffer and embargo the processor, informing the senders.

When a message departs from an emergency-backup buffer, we unembargo the processor, informing the senders.

When a message departs from an assigned buffer, we must find whether any embargoed processors have that buffer assigned to them, and move one of their emergency-backup-buffered messages into the assigned buffer. In that case, we unembargo the processor of the moved message, and inform the senders.

Why do we employ emergency-backup buffers? When a message is placed into a buffer, potentially many processors no longer have any regular buffers available in which to put their messages. We don't even know who those processors are.

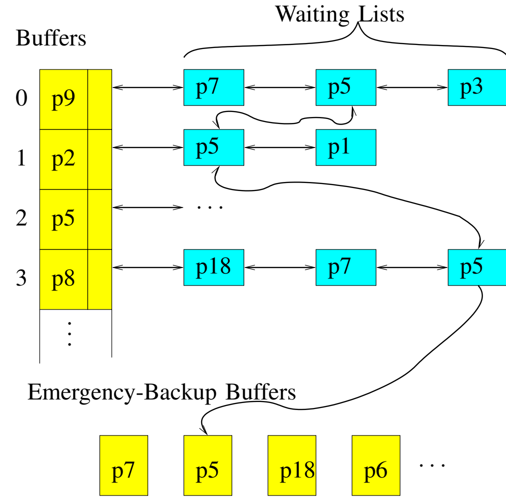


Fig. 1: The bookkeeping data structure comprises buffers, waiting lists, and emergency backup buffers. Each buffer can be empty or hold a message. In the figure, the first four buffers are shown holding messages destined to processors 9, 2, 5, and 8 respectively. Each nonempty buffer also points to a doubly-linked waiting list. For example, the first buffer has a waiting list that includes processors 7, 5, and 3. The double links within a waiting list are shown as horizontal double-ended arrows. Each waiting list element is threaded in a linked list of waiting-list nodes for the same processor. Here we show only the processor links for processor 5, which has a node waiting in buffers 0, 1, and 3. The processor links are shown as double-ended arrows going from one row of waiting list nodes to another. The last waiting list node for a processor points to the emergency backup buffer containing the message for that processor.

So instead of tracking which is a hotspot and which isn't, we accept one extra message from each processor and embargo only the processor to which that message is destined.

The rest of this section shows how to implement this scheme to run in time $O(j)$. The trickiest part is for the receiver to determine what needs to be unembargoed; determining when to embargo a processor is easy, and the senders' jobs are easy too.

The bookkeeping data structure is shown in Figures 1 and 2. For each nonempty buffer, we maintain a pointer to a doubly-linked list, called the waiting list, containing all the identities of all the embargoed processors that could use the buffer. (In the worst case, the expected length of the waiting list is j, since there are at most B embargoed processors, each processor is assigned j buffers, and there are B lists. It turns out we never traverse a waiting list, anyway.) Each waiting list node is also linked in a doubly-linked list of all the waiting list nodes for the same embargoed processor. The last node in that list contains a pointer to the emergency-backup buffer being used by the processor.

Given the bookkeeping data structure, it is straightforward to handle the arrival or departure of a message. When a message arrives, the receiver does the following:

1) Determine if the message can be forwarded directly to an output port (with cut-through routing) (thus achieving goal 2). If so, then the message is sent and no buffering work is performed.

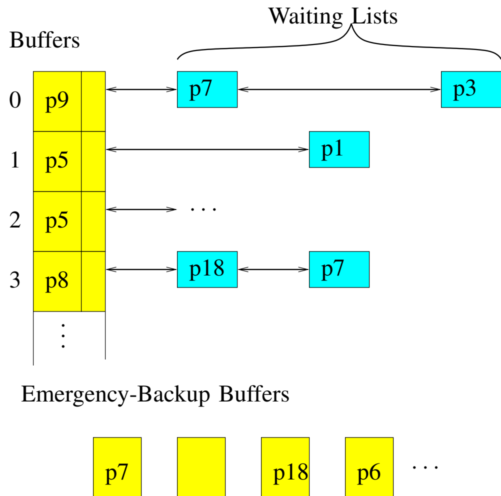


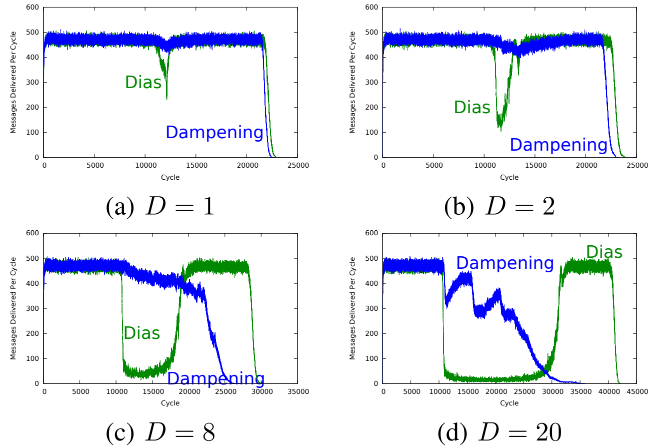
Fig. 2: After removing the message in buffer 1 destined to processor 2 (as shown in Figure 1), we have this state. The message that was in the second emergency-backup buffer has moved to buffer 1, and the waiting lists no longer include processor 5. Processor 5 now has messages in buffers 1 and 2, and is no longer embargoed. The empty emergency-backup buffer has been placed in a linked list (not shown) of all the empty emergency-backup buffers.

2) Otherwise, hash the processor number, to compute the set of buffers that can be used. This is a set of j buffers. Look up each of the j buffers to find a free one.
   a) If there is a free buffer, store the message in the buffer.
   b) Otherwise, store the message in an emergency backup buffer, embargo the processor, and inform the sender of the embargo.
   c) Update the bookkeeping structure, adding the message's processor into the waiting lists of each of the j buffers.
3) If all the buffers fill up, then interdict the senders.

When we clear a buffer by forwarding a message from it to an output port, we must check to see if there is an embargoed message that wants to use that buffer. When forwarding a message from a receiver to its output port do the following:

1) We look at the buffer's waiting list. If that list is nonempty, then we pick an embargoed message (probably we pick the oldest such message to reduce worst-case latency through the switch).
   a) Move the embargoed message into the buffer, freeing the emergency-backup buffer.
   b) Remove the processor from all the waiting lists of all the buffers it wants to use (that's $O(j)$ work, since we have a linked list of the relevant waiting-list nodes).
   c) Inform the sender that the processor is no longer embargoed.
2) Lift the interdiction on the sender, if there is one.

It's not actually necessary to move messages from one buffer to another: it's just bookkeeping to make sure that not too many messages from each processor appear.



(a) $D = 1$ (b) $D = 2$ (c) $D = 8$ (d) $D = 20$

Fig. 3: Simulation results for Dias-Kumar style counting [12] compared to hash dampening. The subfigures show hotspots of various durations D. The horizontal axis of each figure is the cycle time (a message can be fowarded from one switch to another in a single cycle). The vertical axis shows the number of messages delivered per cycle. The hotspot starts after $5,000$ messages has been sent, and can been observed as drop in the number of messages delivered per cycle.

The senders maintain a collection of up to B FIFO's indexed by processor number, to implement virtual output queues which allow it to avoid sending messages destined to particular processors.

Thus for the cost of a constant factor in the number of buffers, we can reduce the flow control information down to at most one stopped or started processor for each message that arrives or departs. Note that due to the latency of communication, a message destined for a hotspot may occasionally arrive even after the processor has been declared as a hotspot by a switch. Our flow control protocol is easy to modify in order take this into account. We can assign the extra message to an emergency-backup buffer; then the processor is only declared as a coolspot once all of the messages it has in emergency-backup buffers have cleared out. (We can have a few extra emergency-backup buffers than we would have had otherwise in order to make up for the fact that some hotspots may be using several.)

Suppose we wanted to employ our scheme in Ethernet. Our flow protocol is not quite compatible with DCE Ethernet. The Ethernet DCE standard provides for flow control, but it provides flow control for only 8 priorities. To make our scheme work, we want flow control for each processor. We don't need to change the Ethernet adapters on the motherboards: it's good enough to do hotspot flow control inside the network, and shut off the whole link from a processor. Since the DCE messages are extendable, it may be acceptable to employ non-standard protocols only between switches, if the motherboards remain unchanged.

IX. Simulation

The main advantage of our dampening switches over the counting scheme of Dias and Kumar [12] is performance. Here

we present some simulation results showing that this advantage can sometimes be substantial.

Figure 3 compares the performance of a hashed dampening switch with a Dias-Kumar style counting switch in which each input port is allowed to hold only one message destined to a given processor. We simulated a store-and-forward butterfly switch containing $1,000$ processors. Each input port of each switch has 13 buffers. The simulation sends 5,000 random messages from each processor, then D messages all to processor zero (creating a hotspot of "duration" D), then 5,000 more random messages. The four subfigures show different hotspot durations. As the figure shows, before the hotspot starts, the two schemes are essentially indistinguishable. For for a relatively short-lived hotspot ($D = 1$) the Dias scheme shows a slight performance drop, and then recovers. As the duration of the hotspot gets longer, the Dias scheme suffers more and more compared to the dampening switch. When $D = 20$, the hotspot is still hurting performance long after the processors have stopped injecting messages into the network, but the dampening switch finishes the job sooner.

X. Caching

If we think of the buffer pool of a switch as a kind of cache, it is interesting to note that we have effectively changed the buffer from a fully associative cache to a randomized j-way associative cache. By reducing the associativity, the behavior of the system is improved against processors that receive too many messages.

One problem that shows up in multithreaded programs is when one particular thread grabs all the cache lines in a shared cache, slowing down other threads. Some approaches to this problem include static partitioning [19]; dynamic partitioning based on identifying program phases [20]; dynamically partitioning to minimize the global hit rate [21], or maximize instructions per cycle [22]. See [19] for a survey of cache partitioning.

In the caching context, a cache line that was brought in by one thread can be evicted by another thread without compromising correctness. One approach is to keep track of which threads are hot and which are cool using one of the schemes described here. A cool thread, when it needs to evict a cache line, can evict any cache line (for example, it might evict the globally least recently used cache line). A hot thread, on the other hand, is allowed to evict only certain cache lines. It might be allowed to evict only cache lines that are were brought in by threads that are at least as hot as it is (when using a counting method), or it might be allowed to evict only cache lines from a certain assigned set of cache lines (when using a hash assignment method).

For caching, there is a clear advantage to allowing a large number of cache lines to be assigned to one thread, whereas in switching, the advantage is second-order.

Furthermore, caching is different from switching in that for caching there are relatively few processors P compared to the number of cache lines B. It would probably make sense to employ the modified counter method described in Section IV, further modified to run in time $O(P)$. The parameters can be adjusted so that, for example, a single uncontended thread can get half the cache.

Acknowledgments

Thanks to Michael Bender who observed that these ideas can be applied to cache competition.

References

[1] G. F. Pfister and V. A. Norton, ""hot spot" contention and combining in multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 943–948, Oct. 1985.

[2] M. Kumar and G. F. Pfister, "The onset of hot spot contention," in *Proc. 1986 International Conference on Parallel Processing*, 1986, pp. 28–34.

[3] A. Bechtolsheim, "Reinventing datacenter networking," in *HPTS*, Asilomar, Pacific Grove, CA, Sep. 2013.

[4] A. Gottlieb, "An overview of the NYU Ultracomputer project," NYU, Ultracomputer Note 100, Jul. 1986. [Online]. Available: https://archive.org/details/overviewofnyuult00gott

[5] A. G. Ranade, "How to emulate shared memory," *J. Comput. Syst. Sci.*, vol. 42, no. 3, pp. 307–326, 1991.

[6] S. L. Scott and G. S. Sohi, "The use of feedback in multiprocessors and its application to tree saturation control," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 4, pp. 385–398, Oct. 1990.

[7] M. Farrens, B. Wetmore, and A. Woodruff, "Alleviation of tree saturation in multistage interconnection networks," in *Proceedings of Supercomputing*, 1991, pp. 400–409.

[8] N.-F. Tzeng, "Alleviating the impact of tree saturation on multistage interconnection network performance," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 107–117, Jun. 1991.

[9] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for VLSI communication switches," in *ISCA*, Honolulu, HI, 1988, pp. 343–354.

[10] R. Merrit, "Bechtolsheim brainstorms on next networking wave," *EE Times*, Oct. 17 2012.

[11] IEEE, "Media access control (MAC) bridges and virtual bridged local area networks — amendment 17: Priority-based flow control," Std 802.1Qbb-2011, Sep. 2011.

[12] D. M. Dias and M. Kumar, "Preventing congestion in multistage networks in the presense of hotspots," in *Proc. 1989 International Conference on Parallel Processing*, vol. 1, Aug. 1989, pp. 9–13.

[13] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the Connection Machine CM-5," *J. Parallel Distrib. Comput.*, vol. 33, no. 2, pp. 145–158, 1996.

[14] A. Borodin and J. E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *Journal of Computer and System Sciences*, vol. 30, pp. 130–145, 1985.

[15] S. J. Swamidass and P. Baldi, "Mathematical correction for fingerprint similarity measures to improve chemical retrieval," *J. Chem. Inf. Model.*, vol. 47, pp. 952–965, 2007, http://www.igb.uci.edu/~pfbaldi/publications/journals/2007/ci600526a.pdf.

[16] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, October 1985.

[17] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[18] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, vol. 3, pp. 267–286, 1979.

[19] D. Sanchez and C. Kozyrakis, "Scalable and efficient fine-grained cache partioning with vantage," *IEEE Micro*, vol. 32, no. 3, pp. 26–37, May 2012.

[20] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proceedings of ISCA'13*, Tel Aviv, 2013.

[21] M. K. Qureshi and Y. N. Patt, "Utility-based cache partioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO 39*, 2006, pp. 432–432.

[22] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr., and J. Emer, "The gradient-based cache partitioning algorithm," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 2012, hIPEAC Papers, Article No. 44.