Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2007-51

Experimental Evaluation of a Coarse-Grained Switch Scheduler

Authors: Charlie Wiseman, Jon Turner, Ken Wong, Brandon Heller

October 19, 2007

Corresponding Author: wiseman@wustl.edu

Abstract: Modern high performance routers rely on sophisticated interconnection networks to meet ever increasing demands on capacity. Regulating the flow of packets through these interconnects is critical to providing good performance, particularly in the presence of extreme traffic patterns that result in sustained overload at output ports. Previous studies have used a combination of analysis and idealized simulations to show that coarse-grained scheduling of traffic flows can be effective in preventing congestion, while ensuring high utilization. In this paper, we study the performance of a coarse-grained scheduler in a real router with a scalable architecture similar to those found in high performance commercial systems. Our results are obtained by taking fine-grained measurements of an operating router that provide a detailed picture of how the scheduling algorithm behaves under a variety of conditions, giving a more complete and realistic understanding of the short time-scale dynamics than previous studies could provide. We also examine computation and communication overheads of our scheduler implementation to assess its resource usage and to provide the basis for an analysis of how the resource usage scales with system size.

Experimental Evaluation of a Coarse-Grained Switch Scheduler

Charlie Wiseman
Washington University
wiseman@wustl.edu

Jon Turner
Washington University
jon.turner@wustl.edu

Ken Wong
Washington University
kenw@arl.wustl.edu

Brandon Heller
Washington University
bdh4@arl.wustl.edu

ABSTRACT

Modern high performance routers rely on sophisticated interconnection networks to meet ever increasing demands on capacity. Regulating the flow of packets through these interconnects is critical to providing good performance, particularly in the presence of extreme traffic patterns that result in sustained overload at output ports. Previous studies have used a combination of analysis and idealized simulations to show that *coarse-grained scheduling* of traffic flows can be effective in preventing congestion, while ensuring high utilization. In this paper, we study the performance of a coarse-grained scheduler in a real router with a scalable architecture similar to those found in high performance commercial systems. Our results are obtained by taking fine-grained measurements of an operating router that provide a detailed picture of how the scheduling algorithm behaves under a variety of conditions, giving a more complete and realistic understanding of the short time-scale dynamics than previous studies could provide. We also examine computation and communication overheads of our scheduler implementation to assess its resource usage and to provide the basis for an analysis of how the resource usage scales with system size.

1.INTRODUCTION

Modern high-end routers such as Cisco's CRS-1 [4] have hundreds or thousands of ports capable of supporting link speeds of 10 Gb/s or more. To ensure scalability, many router architectures utilize internally buffered, multistage interconnection networks that operate with a small speed advantage relative to the external links. While it is easy to design such an interconnect to work well under benign traffic conditions, it is also important for them to operate well under more extreme conditions as the unregulated nature of Internet traffic leads to extreme traffic conditions that can (and do) occur on a fairly routine basis. The key property that an interconnect is expected to have is that it be *nonblocking*, which essentially means that congestion at some output ports should not affect traffic going to other, uncongested outputs. One way to ensure the traffic isolation needed to provide nonblocking performance is to equip the system with a *scheduler* that controls the flow of data through the interconnect with the explicit goal of avoiding internal congestion, while moving traffic through the interconnect as quickly as possible. The challenge in designing such a scheduler is to ensure good performance under extreme traffic conditions for systems that may have tens to thousands of ports, while keeping the resources used for scheduling to a reasonably small fraction of the overall system cost.

Scheduling for large scale routers borrows heavily from the methods that have been developed for smaller systems using crossbar interconnects [1,2,3,8,13]. Crossbar systems are typically limited to a few tens of ports, making it practical to do fine-grained scheduling on a packet-by-packet basis. In such systems, the scheduler matches inputs to outputs during each packet scheduling interval. The crossbar typically operates with a small constant speedup S relative to the external links, which means that it can forward S packets for every 1 packet arrival. Most evaluation of schedulers is limited to theoretical analysis and simulation when the speedup is 2 or more. However, the interface bandwidth between conventional line cards and the interconnect often limits the speedup to be significantly less than 2.

For larger systems, the fine-grained scheduling method used in crossbars becomes impractical, since it is difficult to make scheduling decisions for large numbers of ports in the short time available for scheduling. For example, routers supporting 10 Gb/s links are typically implemented using cell-based interconnects that require between 25 and 40 ns to forward cells. This sets the limit on the time a fine-grained scheduler can spend making scheduling decisions. Clever hardware architectures can make this practical for systems of modest size, but these approaches are not cost effective when scaled up to larger systems.

The use of multistage interconnection networks with internal buffering makes it possible to use a more coarse-grained scheduling approach [10,11]. Instead of attempting to make scheduling decisions on a packet-by-packet basis, coarse-grained scheduling simply regulates the rates at which each input sends to each output over a scheduling interval that is much longer than the individual cell time. Scheduling periods of tens to hundreds of microseconds are reasonable choices since the interconnect can be equipped with sufficient buffering to handle shorter term contention and the delays associated with such coarse-grained scheduling are orders of magnitude smaller than the intrinsic end-to-end delays experienced by packets traveling across wide-area networks.

Coarse-grained scheduling borrows many ideas from crossbar scheduling, such as virtual output queues (VOQs), where each input maintains a separate queue for each output. In addition, many of the scheduling strategies developed for crossbars have natural counterparts in the coarse-grained context. Because coarse-grained schedulers are designed for larger system configurations, they are also typically implemented in a distributed fashion, in contrast to the centralized approach most often used for crossbars. Typically this means that each port has its own *port processor* capable of running some part of the scheduling algorithm and has mechanisms for sharing any necessary information with other ports. The collection of rates chosen by this distributed scheduling process must ensure that no single port is assigned traffic such that it causes the switch to become congested.

In [11], some basic distributed, coarse-grained scheduling algorithms were developed and studied primarily using simulation. [10] included some limited measurement data of a real system, but the measurement methods used required that the system be evaluated at a small fraction of its normal operational speed. That study introduced the idea of using *stress tests* to evaluate the schedulers under extreme traffic conditions.

In this paper, we provide a more comprehensive view of a particular coarse-grained scheduling algorithm. In particular, the contributions of this work include an implementation of one practical scheduler in the extensible routers of the Open Networking Laboratory [7]. A detailed performance characterization of this real implementation is provided. To carry out this evaluation, we developed methods for accurate measurement of switch scheduling algorithms at a time scale that is smaller than the scheduling interval. These mechanisms allow fine-grained examination of the system's dynamic behavior, which leads to a much more precise accounting of the performance impact caused by the coarse granularity of the scheduler. We also present an analysis of the scaling issues associated with distributed scheduler implementations and show how to extend the implementations to make them substantially more scalable.

Section 2 discusses pertinent previous work and then gives the algorithm and implementation details of the scheduler studied in this paper. Section 3 describes our experimental setup, including our traffic generation and synchronization methods and our sub-millisecond measurement agents. Section 4 introduces the traffic pattern we use as a stress test, evaluates the scheduler under the test, and shows that with a moderate speedup the scheduler is able to avoid throughput loss. Section 5 focuses on short time-scale responses to bursty traffic patterns. Section 6 provides a study of the communication and computational characteristics of the scheduler and discusses the scalability of our current implementation along with potential modifications to increase that scalability.

2.BACKGROUND

The coarse-grained scheduler we will be evaluating in this paper is based on the Lowest Occupancy Output First Algorithm (LOOFA) for fine-grained crossbar scheduling originally presented in [6]. The basic tenant of LOOFA is that priority should be given to VOQs for which the associated output queues will empty the soonest. In other words, the goal of the algorithm is to avoid underflow at the outputs by preferentially choosing to send data to those outputs with the lowest output queue occupancy. Each input iterates through the outputs in increasing order of output queue occupancy and requests that it be allowed to send a cell. Outputs grant permission to one input among the requests. This process repeats until no more matches can be made or the time alloted for the scheduling decision is over.

Batch LOOFA (BLOOFA) [11] is the coarse-grained version of LOOFA. As such, the basic idea is the same as LOOFA in that packets in VOQs destined for outputs with the smallest output-side backlog are processed first, but instead of making decisions for individual packets each distributed scheduling component sets rate limits on the VOQs for the duration of the scheduling period. As crossbar scheduling can be reduced to finding matchings in the bipartite graph representation of the crossbar, coarse-grained scheduling can be reduced to finding blocking flows in acyclic flow networks. Existing algorithms for finding such flows can be adapted to produce a centralized implementation of BLOOFA, but for reasons of scalability, a distributed implementation is generally preferred. Distributed BLOOFA (DBL) is a scalable variant of BLOOFA that approximates its behavior, while distributing the relevant computation among the ports. This is accomplished in large part by the use of *backlog-proportional allocation* of VOQ rates based on the total share each port has of all traffic in the system destined to a particular output. To make this precise, we introduce some useful notation before describing the algorithm in detail.

Let $B(i,j)$ be the number of bytes in the VOQ at input i for output j, and $B(j)$ be the total number of bytes in queues at output j awaiting transmission into the link. The notation $B(+,j)$ represents the sum of $B(i,j)$ for all i, that is, the total input-side backlog for output j. Recall that S denotes the speedup, let L be the link capacity in b/s, and let T be the scheduling period in seconds. Finally, let N be the number of ports.

In DBL, each port i sends a message to each port j containing the current value of $B(i,j)$ at the start of the scheduling process. Each port j then sends a message back to every other port containing $B(j)$ and $B(+,j)$. During each scheduling interval, each input can send a total of SLT bits into the switch and each output can receive at most SLT bits. Backlog-proportional allocation is used to decide what share of each output's bandwidth is allocated to each input. Specifically, input i is allowed to send up to $SLT \times B(i,j)/B(+,j)$ bits to output j. Each port allocates its

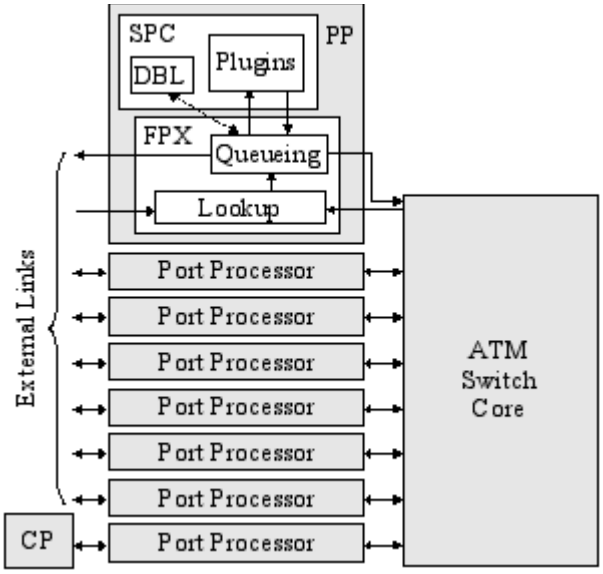

Figure 1. Overview of an ONL router.

own input-side bandwidth by building a list of outputs ordered by increasing values of $B(j)$. Port i traverses this list assigning as much bandwidth as is allowed to each output in the list, until it has allocated all of its input-side bandwidth. The end result of this process is a set of VOQ rates that gives preference to outputs with small output-side backlogs, while ensuring that no output receives enough traffic to cause congestion in the interconnect. Note that the implementation described requires that each port send and receive $2N$ messages.

Evaluation of DBL has until now been limited to simulation, but the results and analytical basis make a good case for further study. As such, it was our choice for an actual implementation. Unfortunately, there are not many options when it comes to choosing a router for our implementation. Most commercially available routers offer no way of extending their functionality in the way we need, and it would be time consuming to build our own router solely to test the scheduler. Instead, we have turned to the Open Network Laboratory (ONL) experimental testbed. A detailed description of ONL can be found at [9], but a brief overview of the relevant architectural features is given here.

The ONL routers are built around a scalable ATM switch core designed to support 1 Gb/s external links and provide an internal bandwidth that is roughly twice the external bandwidth. The ATM core has been augmented with port processors which implement IP route lookup, packet classification, and buffering. As shown in Figure 1, each port processor is composed of a Smart Port Card (SPC) that has an embedded processor for special processing needs and a Field Programmable Port Extender (FPX) that contains a large FPGA configured to handle all the normal packet processing functions. The FPX houses all queueing for the port and provides an interface for the SPC to read queue lengths and set VOQ rates (indicated by the dashed line in Figure 1). The output queues can also be rate controlled to emulate link speeds less than the physical line rate. The general purpose processor available on the SPC is a 500 MHz Pentium III that provides a software plugin environment running on top of a modified NetBSD 3 kernel which allows users to write their own custom code to process packets on the router. It is also responsible for other control and configuration on that port, including any distributed scheduling mechanisms. The SPC kernel is configured to receive clock interrupts every 500 μs, which puts an effective lower bound on the scheduling period.

For the SPC implementation of DBL, the inter-port messages containing backlog information have been modified to take advantage of certain features of the ONL router architecture. In particular, since the ONL routers are equipped with just eight ports, and since the ATM switch core supports a simple multicast mechanism, the DBL implementation sends complete VOQ backlog information and output queue lengths in a single cell that is multicast to all ports. Thus, in each scheduling interval, each port sends one message and receives N. In addition, because the SPCs are at different ports and operate off individual clocks, the DBL implementation in the ONL routers allows each port to run asynchronously with respect to the other ports. Each port maintains a copy of the most recent VOQ and output queue length information received from other ports, and periodically uses this stored information to compute new VOQ rates. The VOQ rates are computed based on the DBL algorithm, with some additional refinements, which we now describe in detail.

At the start of each scheduling interval, the SPC at port i retrieves $B(i,j)$ for all outputs j as well as $B(i)$ from the FPX and sends a single message into the switch with this information. The switch delivers a copy of this message to all of the other ports. Each port uses the most recent information it has received to periodically compute new VOQ rates. This involves computing $B(+,j)$ for all j from the stored VOQ lengths. Outputs are ordered by slightly more complex criteria designed to moderate fluctuations between assigned rates when output backlogs are similar. Specifically, output h comes before output k in the order if and only if one of the following conditions is true:

1. $B(k) - B(h) > C \ (\Leftrightarrow B(h) - B(k) < -C)$
2. (I) $B(h) - B(k) < C$ and

   (II) $B(h) - B(k) < B(+,h) - B(+,k)$

C represents a fixed cutoff value currently set to be $SLT/4$. The intuition behind this is straight forward and most easily explained with the help of Figure 2. If $B(h)$ is enough smaller than $B(k)$ (condition 1), then h comes before k in the order. On the other hand, if $B(h)$ is enough larger than $B(k)$ (condition 2I), then k comes before h. In the region where $B(h)$ and $B(k)$ are close to one another (condition 2II), preference is given to the output with the larger total input-side backlog. Figure 2 illustrates this by highlighting the values of $B(h) - B(k)$ that will result in h

coming before k in the output order, given a particular value of $B(+,h) - B(+,k)$. For example, the middle line in Figure 2 shows an example where condition 2II will determine the output ordering if $B(h)$ and $B(k)$ are within $|C|$ bytes of one another. In all cases, ties are broken by port number with smaller numbers coming before larger ones.

Once the output order has been computed, capacity is assigned to VOQs in order, much as in standard DBL except that a small percentage (around 1% per output) of the input capacity is reserved and assigned to VOQs that would otherwise receive less capacity. This is a small enhancement that leads to better reaction times when previously empty VOQs become active in the middle of the scheduling period. We also replace the pure backlog-proportional bandwidth assignment with a two pass process. This prevents an anomalous characteristic of the DBL algorithm, which can cause an input to assign more bandwidth to one of its VOQs than is needed to clear the entire backlog from that VOQ. When this happens, the input-side bandwidth can be fully allocated to VOQs that do not have much data to send. The first pass limits each VOQ to the minimum of its backlog-proportional allocation and the bandwidth needed to fully clear its backlog within the scheduling interval. In the second pass, inputs that were assigned less than their backlog-proportional allocation in the first pass are allowed to increase their share of the input bandwidth, up to the backlog-proportional allocation. Although some of these changes do make the algorithm more complex, they boost the overall performance of the system for both normal and extreme traffic by smoothing otherwise abrupt behavior and improving reaction time to shifting traffic patterns.

3.EXPERIMENTAL SETUP

In order to evaluate the scheduler in a meaningful way, we needed to be able to take measurements many times per scheduling period. Fortunately, the SPC plugin environment provides a means to do this. Plugins are composed of normal C code that is compiled into a NetBSD kernel module and then loaded onto the SPC to run. Filters can be added to the port that direct packet flows to the plugin for extra processing. For processing not driven by packet arrivals, plugins also have the ability to register callback functions with the kernel, much as the scheduler does.

We have implemented a measurement plugin that is capable of recording VOQ and output queue lengths once every 100 µs, VOQ rates once every 500 µs (that is, once every scheduling period), and packet receive and transmit counts once every 500 µs. All of the measurements are annotated with a timestamp read from the local microsecond-accurate clock. Once the plugin is loaded onto an SPC, it awaits a packet arrival to indicate that it should start taking measurements by registering a callback function that is executed every 500 µs. Each time the callback runs, five queue length measurements are retrieved by calls to the provided plugin API. These measurements are spaced out over the callback period to obtain values every 100 µs.

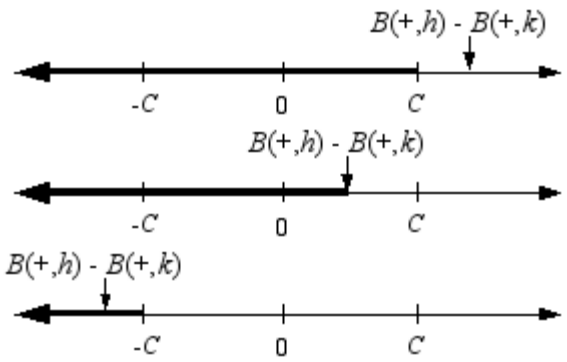

Figure 2. Values of $B(h)$-$B(k)$ for which h comes before k in the output order.

The time between the measurements is filled by a combination of busy-waiting and taking the VOQ and packet rate measurements. Access to the current VOQ rates is provided by an interface to the DBL code, and packet counts are read from the FPX via another plugin API function. The SPC has enough memory to store over 10 s worth of these measurements. After that time, the callback is de-registered and all of the data are sent to the control processor from which the results are easily accessible.

One instance of the plugin is run on each of ports 1 through 7 on the router. Port 0, which is connected to the control processor, is not used as a data port in any of our experiments. To relate results gathered from different plugins, the clocks at different ports must be synchronized to a granularity of less than 100 µs. This is straight forward to do by taking advantage of certain features of the ONL router. We use an existing multicast plugin on port 0 to copy a single UDP packet sent from the control processor to each other port. Each port has a filter setup to direct this special packet to the measurement plugin. Unfortunately, unpredictable start times can still occur if these packets arrive while the scheduler is running because the plugin will not be able to process the packet until the scheduler is finished. To avoid this, we break the process up into two parts. First, the scheduler is deactivated and one packet is sent to each port via the multicast plugin. When that packet arrives at the measurement plugin the current local time is read and recorded as time zero. All subsequent clock reads are adjusted to match this time scale. Second, the scheduler is reactivated and given a short time to stabilize before a another packet is sent that actually begins the measurement process on each plugin. To guard against clock drift these steps were repeated at the beginning of every experiment.

We have verified that when the scheduler is not running and there is no other traffic in the system, the multicast packets will arrive at the plugins on each port within 10 µs of one another. This verification was done by instantiating simple 'echo' plugins on each port other than port 0 and a slightly modified multicast plugin on port 0. A single packet is sent from the control processor to the

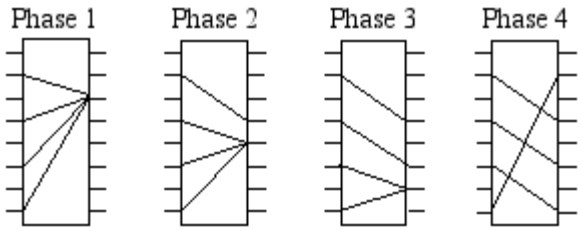

Figure 3. Stress test used with the ONL router.

modified multicast plugin which copies the input packet to each other port as normal. The echo plugins simply re-direct any received packets back to port 0 where they are processed a second time by the modified multicast plugin. For each packet re-arrival, the current time is recorded. The spread of arrival times was never more than 10 μs.

The counterparts to the measurement facilities are the data sources. Generating enough packets to saturate many Gb/s links, particularly with small packets, can be challenging. The brute force approach of aggregating flows from scores of end hosts requires a large investment in machines for testing the capabilities of a single router. Instead, we leverage network processor technology. For some time now, multi-core processors designed specifically with network processing in mind have been available. One example is the Intel IXP chip line [5], which has been included on some PCI-based platforms such as the Radisys ENP-2611 [12]. We have built a flexible packet generation platform on the ENP-2611 that is capable of nearly saturating each of the three Gigabit Ethernet ports on the card with packets of any size.

Each ENP-2611 has a single Intel IXP 2400 on board which contains 8 Micro-Engine (ME) cores for packet processing and one XScale ARM core that acts as the management processor. The high-level design of the packet generator is actually very simple. As is standard for this platform, one ME is dedicated to receiving packets and one to transmitting packets. Three MEs are programmed as timer blocks, where each timer block is responsible for the packets that will leave one port. This is done by preloading the packets that will be sent and configuring each ME with the necessary information to compute inter-packet delays for achieving the desired output rate. Finally, one ME is used to process received packets and potentially start packet flows based on those received packets, as described below. Otherwise, all ME configuration and packet flow control is handled by the XScale.

Three of these packet generators are used in our experimental setup to connect to the seven open (non-control processor) router ports. In order to fully stress the scheduler it is important to be able to start packet flows at each port as closely together as possible. To facilitate this need, the packet generator was built with an option to wait for a packet arrival on one of its interfaces that then serves as the start signal for the ports on that card. There is no general purpose operating system running on the data path of the packet generator and only minimal processing of packets before they reach the synchronization ME, so the time between receiving the packet to acting on the packet is consistent at μs granularity. To synchronize the separate cards, each of the seven used ports on the three cards are connected to a Gigabit Ethernet switch along with another Linux PC. This PC sends a single Ethernet broadcast frame that is placed in a particular VLAN such that it is replicated by the switch once to each packet generator. We have verified that this guarantees that the sources always start within 15 μs of each other by sending packets back to the PC and looking at the first packet arrival times. The seven router ports are also connected to the external switch. VLANs are used again here to ensure correct behavior by mapping each traffic generator port to a single router port and placing them in a unique VLAN. This setup is used for all experiments conducted in this paper.

4.EXTREME TRAFFIC EVALUATION

To study the performance of our implementation of DBL, we start with an extreme traffic pattern. Although there is no known provable worst-case traffic pattern for coarse-grained schedulers, a number of these stress tests have been developed that clearly probe their limits.

This stress test proceeds in M equal length phases and requires M inputs and $2M-1$ unique outputs. During each phase, each input sends constantly to the same output and only switches outputs when the phase changes. In the i-th phase, $M-i+1$ inputs all target one output to overload it and force its total input-side backlog to grow. These outputs are referred to as *overloaded outputs*. After each phase, one input *drops out* of this pattern and begins sending to a new unique output. Once an input has dropped out it continues sending to the same output for the rest of the stress test. These outputs are called *continuous outputs*.

Figure 3 shows the details of the specific instance of the stress test used with the ONL router. During the first phase, all active inputs send to the same output in order to build up a large backlog for that output. In the second phase, one input drops out and the other inputs simultaneously switch to a new output. The scheduler must quickly start servicing the newly active outputs to avoid missed transmission opportunities for those outputs while continuing to ensure that the first output does not underflow. The third phase proceeds similarly with another input dropping out and the other two inputs switching together to a different output. In the final phase the last two inputs each drop out to new unique outputs. This creates an extremely challenging situation for the scheduler to overcome as there are now four continuous outputs that have to be serviced constantly to avoid underflow while simultaneously ensuring that the backlogs for the overloaded outputs are cleared.

This test was designed specifically so that every packet could be transmitted by the end of the final phase. For example, if each phase lasts for 1 second then $4L$ bits will be received during the first phase for the output that is being overloaded and no other data arrive for that output. So

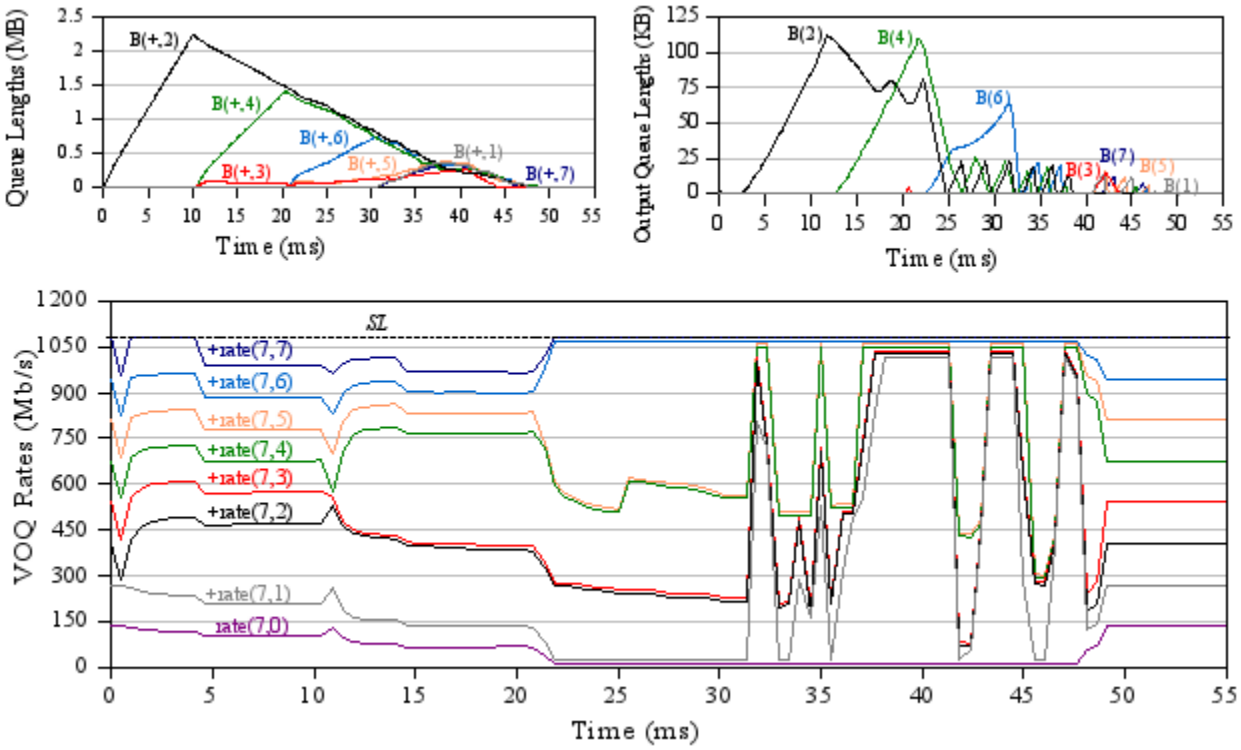

Figure 4. Response for the stress test with *S*=1.2 and *L*=900 Mb/s.

long as the scheduler immediately assigns capacity to those VOQs, there are four full phase times to get those 4*L* bits across the interconnect and thus the output will have 4*L* bits worth of capacity that could be used to send those packets. The situation is similar for all the overloaded outputs. Each of the continuous outputs should be able to finish by the end of the final phase since they are never overloaded. However, if any of the output queues underflow at any time because of contention for interconnect capacity or poor scheduling decisions, then the ideal deadline will be missed. The ratio of actual finish time to ideal finish time is called the overshoot.

Figure 4 shows the performance of our DBL implementation under the stress test in which each phase lasts for 10 ms, link rates are set to 900 Mb/s, and the speedup is 1.2. To set the link rate, the packet generator produces 900 Mb/s of input traffic per link and the ONL router is set to rate limit the traffic leaving each link to 900 Mb/s. The results shown are for 50 byte UDP payload packets, but the behavior is nearly identical when the same test is run for packets of any size. The upper-left graph shows the total input-side backlogs for each output. Each of the overloaded outputs (i.e., ports 2, 4, and 6) build backlogs as expected. The continuous outputs in phases 2 and 3 (ports 3 and 5) only build minuscule backlogs until the final phase begins. At that point, however, all the continuous outputs begin building significant input-side backlogs, which results in an overshoot of about 20%.

The bottom chart of Figure 4 shows the VOQ rates for port 7 in cumulative form, i.e., +*rate*(7,*j*) represents the sum of the VOQ rates at input 7 for outputs 0 through *j*. The most striking feature is the extreme rate fluctuation during the final phase of the test. Once the final phase begins, the scheduler has to keep port 1 supplied with a steady stream of packets while still clearing each of the three input-side backlogs. By time 33, all of the output-side backlogs have become small enough that they empty within one scheduling period unless supplied with new packets. The end result is that the scheduler enters a relatively unstable state and tends to direct most of the capacity to only one or two outputs during each scheduling period. A similar test with a speedup of 1.5 also exhibits rate fluctuations, but they are much less erratic.

A curious phenomenon exists around time 5 when all of the rates except *rate*(7,2) suddenly decrease. To understand why this occurs, consider the output queue length at port 2 at that time, as shown in the upper right chart of Figure 4. Recall that the output ordering scheme gives preference to outputs with larger input-side backlogs when the output-side backlogs are similar. At time 5, the output backlog for port 2 grows past that cutoff value and moves from the front of the order to the back of the order. The second phase of VOQ rate assignments attempts to share any left over input-side capacity fairly among each VOQ based on the same output ordering. In this case, the VOQ for output 2 is limited by output-side capacity and can not take its share of the extra input-side capacity. As output 2 is now at the end of the output order, that capacity remains

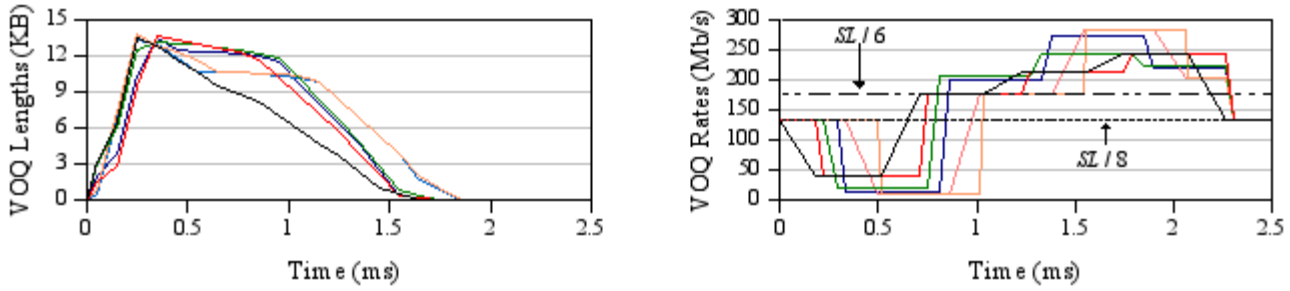

Figure 5. Response to a sudden burst of traffic.

unused. This also explains the feature at time 15, and more generally why all of the input-side capacity is not always assigned. Multiple extra passes could be made to continue assigning the extra capacity until it is all taken or some fixed number of passes has elapsed. Perhaps a better solution would be to re-order the outputs for the second pass based on the amount of extra capacity that they could use before reaching their output-side limits.

Another interesting feature occurs at time 25 when rate(7,4) suddenly jumps by over 100 Mb/s. During this third phase, port 7 has traffic for each of the three overloaded outputs. Based only on its share of the output-side capacities for those ports, port 2 would get one quarter of the input capacity, port 4 would get one third, and port 6 would get one half. The result is that port 7 becomes input capacity constrained and so the output order determines which of the three VOQs will not get its desired rate. By examining the input and output backlog charts, we see that at the beginning of the third phase the order of the overloaded outputs is 6,2,4. This follows because 6 has a much smaller output backlog than the others and for outputs with similar output backlogs and total input-side backlogs, ties are broken by lower port number. Then at time 25, the output backlog for 6 becomes much larger than the others which changes the output order to be 2,4,6. Thus port 4 is able to get its desired share while port 6 is not.

To see how our implementation of DBL performs with varying values of speedup, we included a run-time configuration option in the scheduler that allows the speedup to be changed. While this does not change the actual capacity of the switch interconnect, it effectively does so by limiting the total rate at each input and the total rate destined for each output to be no more than *SL*. So long as *SL* is kept below the actual switch capacity then the system will behave as expected, i.e., there will be no congestion in the interconnect. The curve labeled 'stress test' in Figure 6 shows the overshoot as the speedup varies between 1 and 1.6. For speedups of 1.5 or larger, the overshoot becomes negligible.

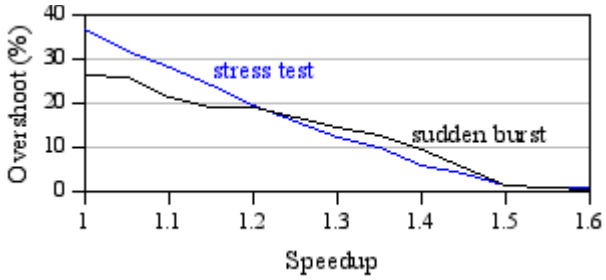

Figure 6. Overshoot of various traffic patterns.

5.DYNAMIC TRAFFIC EVALUATION

It is also important to study the behavior of our DBL implementation under more realistic, but still demanding, traffic patterns. To that end, we now focus on responses to two types of bursty traffic.

The first test is a simple one designed to examine the short time-scale response to a sudden burst of traffic from all inputs destined for the same output. In particular, ports 2 through 7 simultaneously receive a small number of 50 byte UDP payload packets destined for port 1. Although burst lengths between 100 µs and 500 µs were tested, a burst length of 250 µs most often resulted in the largest overshoot primarily due to the asynchronous operation of the scheduler. Figure 5 shows the scheduler's response to this traffic pattern. The right chart shows how the VOQ rates change over the course of the burst. Ideally, the rates would all start and end at *SL*/8 Mb/s (the nominal rate when the system is idle) and rise to *SL*/6 Mb/s (each input equally sharing the capacity to output 1) until the burst was finished. Instead, as the scheduler on each port runs for the first time since the burst began, the rates are set to near zero for one scheduling interval before rising to the expected rate. This is actually due to a mechanism in our implementation for safe asynchronous operation where each port delays using its own current queue length measurements for one scheduling interval (it still sends the current values to the other ports). This forces the scheduler to wait until every port has had a chance to send queue length updates. To understand the necessity for such behavior, consider what might happen otherwise. The first port that runs the scheduler when traffic arrives for a previously idle output will believe that is has the entire input-side backlog for that output and could potentially assign the entire switch capacity to that VOQ. Now consider that all of the ports might run the scheduler within a few µs of one another. If a burst arrived at every input for one output, that output would be assigned *N* times its capacity! The safety mechanism ensures that each port will see the size of the backlogs from each other port before reacting. As Figure 5 shows, however, this mechanism has an unintended draw-

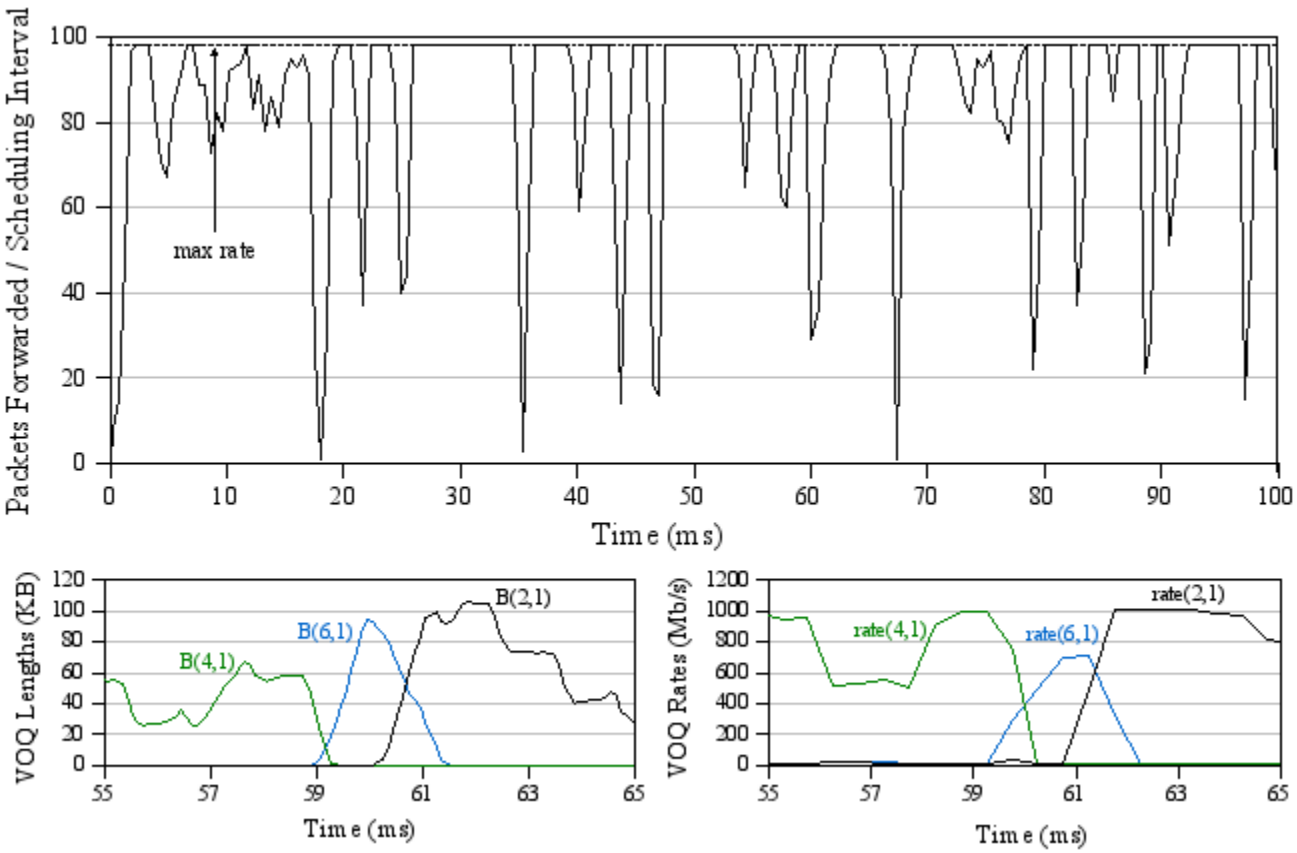

Figure 7. Response for a bursty traffic pattern.

back. Once the input-side backlogs begin to clear, this delayed reaction causes each input to believe that it now has a larger input backlog for the output than it actually does. This results in VOQ rates being set too high, and the output is ultimately over-subscribed before settling back to the nominal rate. It is worth noting that this behavior would be nearly impossible to detect without our fine-grained measurement infrastructure.

The same test was run for a range of speedups from 1 to 1.6, where each test consisted of 20 trials for that speedup value. The maximum overshoot is displayed as the 'sudden burst' curve in Figure 6. The results are similar to those from the stress test with the notable exception that performance under the sudden burst test is actually worse for mid-range speedups.

The second bursty traffic pattern we use is similar to one from [11]. One output, the *subject*, is chosen to receive a steady stream of packets at the line rate during the entire test. The input that is currently sending to the subject changes randomly over the course of the test, sending for an exponentially distributed time before a new input is selected. The other inputs each independently choose an output other than the subject and continue sending to that output for an exponentially distributed amount of time before moving to a new output. This process leads to roughly one quarter of the outputs being overloaded at any given time. Ideally, the scheduler will supply the subject with a steady stream of packets over the entire test in order to keep the output link busy, but contention at the input-side of the interconnect can result in bursty output traffic.

The behavior of DBL for L=900 Mb/s and S=1.2 with 500 bytes UDP payload packets is shown in Figure 7. Port 1 is the subject output. The top chart shows the number of packets forwarded by the subject every scheduling interval. A link rate of 900 Mb/s leads to a maximum rate of 98 packets per interval as indicated by the dashed line. As expected, the line rate is not always attained. Indeed, for nearly 60% of the test input-side contention results in the forwarding rate being less than the line rate. A more complete picture is given from time 55 to 65 in the bottom charts. During this time interval, port 4 sends to the subject until time 59 when port 6 begins sending to it for only 1 ms. At time 60, port 2 takes over sending to the subject. The bottom left chart shows these inputs' VOQ lengths for the subject and the bottom right chart shows the associated VOQ rates. The scheduler can not keep the subject supplied with 900 Mb/s of traffic during most of this period, primarily due to the rapid changes in traffic that occur over only two scheduling intervals.

6.SCALING CONSIDERATIONS

It is important to understand how the resources required for distributed scheduling change as the system size scales

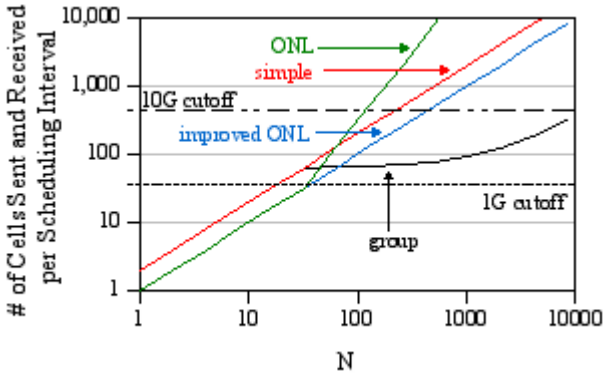

Figure 8. Overhead of various communication methods.

up. First, we address the communication overhead by considering the method used to share the required information among the N ports. In the approach outlined in the original DBL description, which we will refer to as *simple*, each port i starts by sending a message to every port j containing $B(i,j)$. Each port j then computes the sum $B(+,j)$ and sends a message to every other port containing $B(+,j)$ and $B(j)$. This implies that every port sends $2N$ cells and receives $2N$ cells for control purposes. In order to keep the overhead to a small fraction of the total capacity, the scheduling period must be substantially larger than the time it takes to send $2N$ cells to the switch. For the ONL switch, the time to send a cell is about 250 ns. To limit the overhead for scheduling to 2% of the switch bandwidth, then, we need a scheduling period of at least $12.5N$ µs. This implies that the largest acceptable value of N is 40 with our standard 500 µs scheduling period.

The method used in the ONL routers is somewhat different than the simple method. Because the ONL routers are small, each port can place all of its VOQ lengths plus the output queue length into a single cell which is multicast to the other ports. This gives each port all the information it needs to make its scheduling decisions. Here, each port sends one cell and receives N. This approach can be used directly so long as N is no larger than the number of VOQ values that can be carried in a single cell. Denote this value by k and note that if VOQ lengths are represented using a simple floating point representation, we can comfortably handle values of k up to 32. One can scale this approach directly to larger systems by sending messages consisting of multiple cells and multicasting them in the same way. In this approach, called *ONL*, each port sends $\lceil N/k \rceil$ cells and receives $\lceil N/k \rceil N$. Note that this implies a quadratic dependence on the switch size, which greatly limits its ability to scale. However, simply creating separate multicast channels for each group of k consecutive ports can reduce the number of cells received back to N. We refer to this scheme as *improved ONL*.

To further improve the scalability, a grouping technique can be used to aggregate partial sums that are then used to obtain the $B(+,j)$ values. The ports are divided into groups of h consecutive ports where $hk \geq N$. Within each group, each port sends its first k VOQ lengths to the first member of the group, the next k VOQ lengths to the next member, and so forth. Each group member can use the VOQ values received to compute a partial sum. In the next phase of the process, the i-th member of each group exchanges all of the partial sums it has computed with the i-th member of every other group. It then uses these to compute the $B(+,j)$ values for outputs j in the range $[(i-1)k, ik-1]$. Finally, these values are sent to the other members of its group. This method, called *group*, requires each port to send and receive $2\lceil h/k \rceil h+(\lceil N/h \rceil-1)$ cells. Note that it does not require the use of multicast.

Figure 8 shows how these methods compare with one another as the switch size grows, assuming $k=h=32$. The two cutoff lines correspond to points above which the communication overhead for distributed scheduling exceeds 2% of the switch bandwidth. The line labeled 1G is computed based on the parameters for the ONL router (250 ns cell time and a 500 µs scheduling interval). The line labeled 10G assumes that the cell time scales down by a factor of 10 while the scheduling interval is held constant. Of course, the positions of the cutoffs change as one adjusts the acceptable overhead and/or the scheduling interval. For an ONL-like switch, no approach scales to more than a few tens of ports without the communication overhead exceeding 2% of the switch capacity. However, the group method allows 10G switches to scale up to thousands of ports while keeping the overhead below 2%. Even for the 1G case, the group method does scale up to 1,000 ports if a communication overhead of 5% is acceptable.

The other aspect of distributed scheduling that affects scalability is the computational overhead. We annotated the scheduler with timing code in order to understand the actual computational needs of our implementation of DBL. Results were gathered over 100,000 iterations of the scheduler and the maximum run times recorded. There are three basic components to the scheduler: reading queue lengths from the FPX, sending and receiving queue lengths cells, and computing the new VOQ rates. Reading queue lengths from the FPX took just under 15,000 cycles (30 µs on the SPC). For the majority of those cycles, the SPC is merely waiting on the response from the FPX. The current interface only allows for synchronous communication with the FPX, but if an asynchronous model existed then the SPC could potentially be doing other work while it waited on the response. The actual act of sending and receiving the queue length cells takes less than 2,500 cycles (5 µs). Computing the VOQ rates takes the most computational resources, needing just over 20,000 cycles (40 µs). This is largely dominated by the computation of the $B(+,j)$ values for each port.

The group method outlined earlier is preferable from a computational standpoint. Both variants of the method used in the ONL routers require that each port processor perform $(N-1)N$ additions to compute the $B(+,j)$ values.

The group method requires only $(h-1)k+(\lceil N/h \rceil-1)k$ additions. As an example, for N=1024 and k=h=32 this is 1,984 additions versus over 1 million for the ONL method. Of course, in the ONL routers N is small enough that the inefficient procedure for computing the sums is not a significant concern, but clearly it can become a limiting factor as N increases. The 500 MHz processor used in the ONL router can perform 2,000 additions in under 35 μs, so the computational effort required to calculate the sums in the group method is not a serious constraint.

The method for ordering the outputs will also become more important as N scales up. Once again, the computational effort to do this on an ONL router with only 8 ports is minimal. Any simple sorting routine will work quickly enough, but moving to larger routers would necessitate a more sophisticated algorithm. Certainly, the fact that the output order is relatively stable over time could be leveraged to achieve better performance.

7.CLOSING REMARKS

Current high-end routers are being built with buffered, multistage interconnection networks. While these interconnects offer one of the most scalable and cost-effective solutions for such routers, it is known that their performance degrades significantly under certain traffic conditions that do occur in the Internet. Moreover, the interconnect speedup is often much smaller than 2, which means that the analytical performance guarantees of most general schedulers do not apply. Distributed, coarse-grained schedulers have the potential to fill the gap. Until now, all serious evaluation of this class of schedulers has been limited to either simulation or analysis that requires a speedup of 2. We have presented a real implementation and evaluation under much more realistic conditions. In particular, we have studied the performance of one particular scheduler under both extreme and dynamic traffic patterns at Gb/s rates when the speedup is considerably lower than that required by the analytical results. We have also given some insight into the actual scalability of distributed, coarse-grained schedulers based on our experience implementing one in a real router.

We have also developed a measurement infrastructure that is capable of producing meaningful results at a 100 μs granularity. This infrastructure has allowed us to evaluate the fine-grained responsiveness of our scheduler in a way previously impossible on real running systems. Coordinated measurement agents running on each port of the router examine queue lengths, VOQ rates, and packet input and output rates so that the fine-grained behavior of the scheduler can be observed. Packet generation mechanisms comprise the rest of the measurement infrastructure. We have developed a flexible packet generator that runs on a network processor contained on a PCI card that is able to produce up to three Gb/s of any size packets. Additionally, it is possible to synchronize multiple packet generators such that the generators start sending packets within 15 μs of each other. This infrastructure was invaluable in both the debugging and evaluation phases of our study, and we believe that it can be extremely useful for many kinds of network evaluation involving fine-grained dynamics.

We worked with the Open Network Laboratory experimental testbed for all our evaluation in this paper. While much of our work was able to proceed within the normal ONL environment, implementing the scheduler in the core of the ONL router did require some extra privileges and attention from the ONL staff. We are also currently working with the ONL staff to add our packet generators to the standard ONL hardware suite.

References

[1] Anderson, T., S. Owicki., J. Saxe and C. Thacker. "High Speed Switch Scheduling for Local Area Networks," *ACM Trans. on Computer Systems*, 11/93.

[2] Chuang, S.-T., A. Goel, N. McKeown, and B. Prabhakar "Matching Output Queueing with a Combined Input Output Queued Switch," *IEEE Journal on Selected Areas in Communications*, 12/99.

[3] Chuang, S.-T., S. Iyer, and N. McKeown. "Practical Algorithms for Performance Guarantees in Buffered Crossbars," *Proceedings of IEEE INFOCOM*, 3/05.

[4] Cisco Carrier Routing System. http://www.cisco.com/en/US/products/ps5763/index.html.

[5] Intel IXP 2xxx Product Line of Network Processors. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[6] Krishna, P., N. Patel, A. Charny and R. Simcoe. "On the Speedup Required for Work-conserving Crossbar Switches," *IEEE J. Selected Areas of Communications*, 6/99.

[7] Kuhns, F., J. Dehart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, W. D. Richard, D. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, and K. Wong. "Design and Evaluation of a High-Performance Dynamically Extensible Router," *Proceedings of the DARPA Active Networks Conference and Exposition*, 5/02.

[8] McKeown, N. "iSLIP: A Scheduling Algorithm for Input-queued Switches," *IEEE Transactions on Networking*, 4/99.

[9] Open Network Laboratory. http://www.onl.wustl.edu.

[10] Pappu, P., J. Parwatikar, J. Turner and K. Wong. "Distributed Queueing in Scalable High Performance Routers," *Proceedings of IEEE INFOCOM*, 4/03.

[11] Pappu, P., J. Turner and K. Wong. "Work-Conserving Distributed Schedulers for Terabit Routers," *Proceedings of SIGCOMM*, 9/04.

[12] Radisys Products. http://www.radisys.com/products/OtherProducts.cfm.

[13] Rodeheffer, T. L., and J. B. Saxe. "An Efficient Matching Algorithm for a High-Throughput, Low-Latency Data Switch," Compaq Systems Research Center, Research Report 162, 11/5/98.