Packet Scheduling in a Low-Latency Optical Interconnect with Electronic Buffers

Lin Liu
Dept. Electrical & Computer Engineering
Stony Brook University
Stony Brook, NY 11794, USA

Zhenghao Zhang
Computer Science Department
Florida State University
Tallahassee, FL 30306, USA

Yuanyuan Yang
Dept. Electrical & Computer Engineering
Stony Brook University
Stony Brook, NY 11794, USA

Abstract

Optical interconnect architectures with electronic buffers have been proposed as a promising candidate for future high speed interconnections . Out of these architectures, the *OpCut* switch [1] achieves low latency and minimizes optical-electronic-optical (O/E/O) conversions by allowing packets to cut-through the switch whenever possible. In an OpCut switch, a packet is converted and sent to the recirculating electronic buffers only if it cannot be directly routed to the switch output. In this paper, we study packet scheduling in the OpCut switch, aiming to achieve overall low packet latency while maintaining packet order. We first decompose the scheduling problem into three modules and present a basic scheduler with satisfactory performance. To relax the time constraint on computing a schedule and improve system throughput, we further propose a mechanism to pipeline packet scheduling in the OpCut switch by distributing the scheduling task to multiple "sub-schedulers." An adaptive pipelining scheme is also proposed to minimize the extra delay introduced by pipelining. Our simulation results show that the OpCut switch with the proposed scheduling algorithms achieve close performance to the ideal output-queued (OQ) switch in terms of packet latency, and that the pipelined mechanism is effective in reducing scheduler complexity and improving throughput.

Index Terms: Optical interconnect, packet scheduling, pipelined algorithm.

I. Introduction

In recent years, interconnects draw increasingly more attention due to the fact that they tend to become a bottleneck at all levels: intra-chip, chip-to-chip, board level, and computer networks. There are many requirements posed on an interconnect, such as low latency, high throughput, low error rate, low power consumption, as well as scalability. Finding a solution that can satisfy all these needs is a non-trivial task.

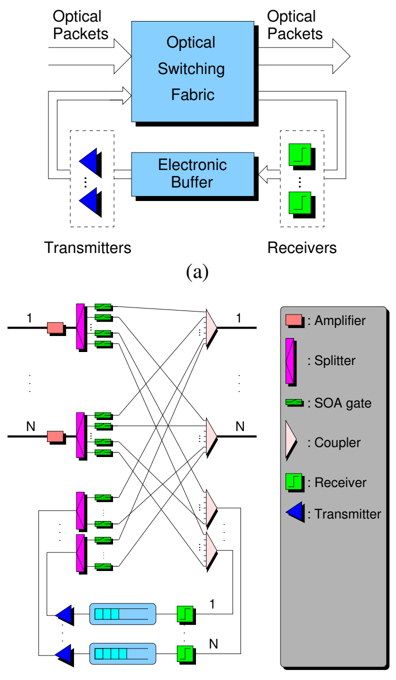
Optical fibers, featured with high bandwidth and low error rate, are widely recognized as the ideal media for interconnects. Some optical interconnect prototypes have been built and exhibited, for example, the recent PERCS (Productive, Easy-to-use, Reliable Computing System) project [14] and OSMOSIS (Optical Shared Memory Supercomputer Interconnect System) project [3][4][5] at IBM. It has gradually become consensus that future high speed interconnects should exploit as many advantages optics can provide as possible.

One of current major hurdles in optical interconnect design is the absence of a component equivalent to the electronic random access memory, or optical RAM. Currently the most common approach to buffering in optical domain is by letting the signal go through an extra segment of fiber, namely, the "fiber delay line" (FDL). An FDL generates a fixed buffering delay for any optical signal, which is in fact the propagation delay for the signal to transfer over the FDL. To provide flexible delays, FDLs have to be combined with switches. Extensive research has been devoted to the realization of large, powerful all-optical buffers [9] [10] [11] but the random accessibility is still absent.

Alternatively, there are emerging techniques that aim at slowing the light down, for example, [7][8]. While these researches present interesting results towards implementing optical buffers with continuous delay, so far it is still unclear whether slow light can provide sufficiently large bandwidth and buffering capacity for it to be used in practical systems. Therefore, currently electronic buffer seems to be the only feasible option to provide practical buffering capacities.

Another core component of an interconnect is the packet scheduler. The scheduling process introduces extra delay because the scheduling algorithm may need time to converge. In addition, as the scheduler is usually located at the center of the switch, if an input port must receive a grant from the scheduler before sending packets, the round-trip delay between the input port and the scheduler becomes significant in low latency applications. For example, it was estimated in [2], [5] that the round-trip delay can be as long as 64 time slots, where a time slot is about 50ns which is the time needed to transmit a packet.

To address these challenges, a low latency switching architecture that combines optical switching with recirculating electronic buffer was recently proposed in [1]. In the following, we simply refer to it as the *OpCut* (Optical Cut-through) switch. Fig. 1(a) shows a high-level view of the switch. The key feature of the OpCut switch is that the arrived optical packets will be routed to the output directly, or "cut through" the switch, whenever possible. Only those that cannot cut through are sent to the receivers, converted to electronic signals and buffered, which can be sent to the output ports later by the optical transmitters. By allowing packets to cut through, the OpCut switch significantly reduces both packet latency and O/E/O conversions. Also, the OpCut switch does not hold any packet at the input ports. It always sends packets to the switching fabric, because with the number of receivers equal to the number of input ports, there can always be found a receiver to "pick up" a packet that needs to be buffered. Hence, the OpCut architecture eliminates



(b)

Fig. 1. (a) A high level view of the OpCut switch. (b) A possible implementation of the OpCut switch.

the round-trip delay between the input ports and the scheduler. Due to these reasons, the OpCut switch architecture holds the potential of achieving very low latency. **In addition, some recent research, for example, [6], also shows that such hybrid optical packet switching with shared electronic buffer could lead to substantial saving in power and cost over all-optical or electrical technologies.**

The feasibility and the performance of the OpCut architecture was studied in [1] under a simple random packet scheduling algorithm. While the random algorithm is convenient for analytical modeling, it is impractical because it is difficult to implement random functions at high speed[16]. In addition, the random scheduling algorithm cannot maintain packet order, which is generally desired for switches [21], [13].

There has been a lot research on scheduling algorithms in packet switches. One of the most extensively studied topics is packet scheduling algorithms for the input-queued (IQ) switch. The IQ switch is usually assumed to work in a time-slotted fashion. Packets are buffered at each input port according to their destined output port, or in virtual output queues (VOQ). The scheduling problem for a time slot is formalized as a bipartite matching problem between the input ports and the output ports of the switch. Existing scheduling algorithms for IQ switches can be divided into two categories: *maximum weighted matching* based optimal algorithms and *maximal sized matching* based fast algorithms. The first category includes algorithms such as Longest Queue First (LQF) and Oldest Cell First (OCF) [12]. These algorithms have impractically high computing complexity, but are of theoretical importance as they deliver 100% throughput under virtually any admissible traffic. The second category includes, for example, Round Robin Greedy Scheduling (RRGS) [17], Parallel Iterative Matching (PIM) [15] and iSLIP [16]. These algorithms only look for a maximal matching in each time slot hence have practical time complexity. They are therefore preferred in real systems, although they can only give sub-optimal schedules.

However, in a high speed or ultra high speed system, even these maximal matching scheduling algorithms may become a potential problem. As the length of a time slot shrinks with the increase in line card rate, it may become too stringent to calculate a schedule in each single time slot. In such a case, pipelining scheduling is needed to relax the time constraint. In [17] a pipelined version of the RRGS algorithm was reported, in which each input port is assigned a scheduler. The scheduler of an input port selects an idle output port, and passes the result to the next input port. The process goes on until all input ports have been visited. However, this approach introduces an extra delay equal to the switch size. In [18] the pipelined maximal-sized matching algorithm (PMM) was proposed, which employs multiple identical schedulers. Each of these schedulers independently works towards a schedule for a future time slot. As pointed out in [19], PMM is "more a parallelization than a pipeline" since there is no information exchange between schedulers. [19] further proposed to pipeline the iterations of iterative matching algorithms such as PIM and iSLIP by adopting multiple sub-schedulers, each of which taking care of one single iteration and passing the intermediate result to the next sub-scheduler in the pipeline. One problem with this approach is that it may generate grants for transmission to an empty VOQ since at any time a sub-scheduler has no idea about the progress at other sub-schedulers and may try to schedule a packet that has already been scheduled by other sub-schedulers. As a result, the service a VOQ receives may exceed its actual needs and is wasted.

In this paper, we will study the packet scheduling problem in the OpCut switch. The rest of this paper is organized as follows. Section II introduces the OpCut switch architecture. Section III presents the basic packet scheduler for the OpCut switch. Section IV proposes a mechanism to pipeline the scheduling procedure, including an adaptive pipelining scheme. Section V presents the simulation results. Section VI compares the OpCut switch with some recently proposed architectures. Finally Section VII concludes the paper.

II. The OpCut Switch

In this section we briefly describe the architecture of the OpCut switch. The OpCut architecture may adopt any switching fabric that provides non-blocking connections from the inputs to the outputs. One possible switching fabric is shown in Fig. 1(b). It has N input fibers and N output fibers, numbered from 1 to N, and there is one wavelength on each fiber. In the following, we interchangeably use input (output) fiber, input (output) port, and input (output). Each input fiber is sent to an amplifier. The amplified signal is broadcast to N output fibers under the control of SOA gates. In addition, each signal is also broadcast to N receivers. An output fiber receives the signal and routes it to the processor or the next stage switch. A receiver converts the optical packet into electronic forms and stores it in its buffer. There is one transmitter per receiver buffer. A transmitter can read the packets and broadcast the selected packet to the output fibers, also under the control of SOA gates, such that the packets in the buffer may be sent out.

In this paper, we follow the same assumptions as other optical switch designs that the switch works in time slots, and packets of fixed length fit in exactly a time slot. The length of a time slot is about 50 ns, similar to that in the OSMOSIS switch [2], [5]. Before receiving the packets, the switch is informed about the destinations of the packets. This can be achieved, for example, by letting a group of processors share an electronic connection to send the packet headers to the switch. The headers are sent to the switch before the packet is sent to allow the switch to make routing decisions and to configure the connections. Note that the cost associated with this electronic connection is likely to be small, because the link is shared by multiple processors and the link speed can be much lower than the data link. At the beginning of each time slot, up to one packet may arrive in optics at each input port. We define a flow as the stream of packets from the same input port and destined for the same output port. Unlike other optical switches with electronic buffers in which every packet goes through the O/E/O conversions, in the OpCut switch packets are converted between optics and electronics only when necessary. Whenever possible, arrived optical packets are directly sent to their desired output port, or, cut-through the switch. A packet that cannot cut-through is picked up by one of the receivers and sent to the electronic buffer. Later when its destined output port is not busy, the packet can be fetched from the electronic buffer, converted back into optics, and scheduled to the switch output.

In each time slot, a receiver picks up at most one packet, and a transmitter sends out at most one packet. In other words, there is no speedup requirement. The cost of the OpCut switch is mainly determined by the number of transmitters, the number of receivers, and the size of the switching fabric. It can be seen that the OpCut switch needs N transmitters and N receivers. The switch fabric is $N \times 2N + N \times N$, where the $N \times 2N$ part connects the inputs to the outputs and receivers, while the $N \times N$ part connects the transmitters to the outputs. To connect more processors, multiple OpCut switches can be connected according to a certain topologies, similar to the Infiniband switch.

III. The Basic Packet Scheduler for the OpCut Switch

The key challenge to achieve low latency in a switch is the design of the packet scheduling algorithm. Due to its feed-back buffer structure, existing scheduling algorithms cannot be directly applied to the OpCut switch. Keeping packet order also becomes more challenging in the OpCut switch. For example, in input-queued switches, packets belonging to the same flow are stored in the same Virtual Output Queue (VOQ), thus packet order is preserved as long as each VOQ works as a FIFO. In the OpCut switch, however, packets from the same flow may be picked up by different receivers.

The scheduling algorithm for an OpCut switch should give answers to the following three questions:

- Question 1. For the newly arrived packets, whether they may go to the output port directly or they should be buffered.
- Question 2. For a packet that should be buffered, which receiver should be used to pick it up.
- Question 3. For the output ports that are not receiving the new packets, which of the buffered packets may be sent to the output port.

This section is organized around how the three questions can be answered. We start with the notations and the basics of the scheduler.

A. Notations and Basics of the Scheduler

In an OpCut switch, input i is denoted as I_i, output j is denoted as O_j, and receiver r is denoted as R_r. Flow ij is defined as the stream of packets arrived at I_i destined to O_j. We also refer to the time slot in which a packet arrives at the switch input as the *timestamp* of that packet. Among all packets of a flow currently at the switch input or in the buffer, the one with the oldest timestamp is referred to as the *head-of-flow* packet. Maintaining packet order means that a packet must be a head-of-flow packet at the instant it is being transmitted to the switch output.

The OpCut scheduler adopts round-robin scheduling when the scheduler has multiple choices and has to choose one. Similar to [16], the round-robin scheduler takes a binary vector as input, and maintains a pointer to make the decisions. Let $[r_1, r_2, \ldots, r_N]$ be the input binary vector and let g be the current round-robin pointer g where $1 \leq g \leq N$. The scheduler picks the *highest priority element* defined as the first '1' encountered when searching the elements in the vector from r_g in an ascending order of the indices, while wrapping around back to r_1 when reaching r_N. Incrementing the round-robin pointer g by one beyond x means that $g \leftarrow x + 1$ if $x < N$ and $g \leftarrow 1$ if $x = N$.

B. Queueing Management

For each output port, the scheduler of the OpCut switch keeps the information of the packets that are in the buffer and are destined to the output port in a "virtual input queue" (VIQ) style. Basically, for output O_j, the scheduler maintains N queues denoted as F_{ij} for $1 \leq i \leq N$. For each packet arrived at I_i destined for O_j and are currently being buffered, F_{ij} maintains its timestamp, as well as the index of the buffer the packet is in. Note that F_{ij} does not hold the actual packets.

Packets are stored at the receiver buffers. It would make the scheduling much easier if each receiver maintains a dedicated queue for each flow. However, this will result in N^3 queues over all receivers and will lead to much higher cost which is unlikely to be practical when N is large. Instead, no queue is maintained in any receiver buffer, and an auxiliary array is adopted in each buffer to facilitate the locating of a specific packet. The auxiliary array is indexed by (partial) timestamps to store the location of packets in the buffer. Since in each time slot a receiver picks up at most one packet, it is able to locate a packet in constant time given the timestamp of the packet and the auxiliary array. Note that some elements of the array may be empty but the packets can always be stored continuously in the buffer.

As an implementation detail, the auxiliary array can be used in a wrap-around fashion thus does not need to have very large capacity. For instance, if the index of the array is 8-bit long, then the array stores the location of up to 256 packets. Consequently, only the lower 8 bits of the timestamp is needed to locate a packet in the buffer. A conflict occurs only if a packet is routed to a receiver, and another packet picked up by the same receiver at least 256 time-slots earlier is still in the buffer. When this is the case, it usually indicates heavy congestion. Hence it is reasonable to discard one of the packets.

C. The Basic Scheduling Algorithm

Next we describe a basic scheduling algorithm for the OpCut switch. To maintain packet order, the basic algorithm adopts a simple strategy. Basically, it *allows a packet to be sent to an output only if this packet is a head-of-flow packet*. The basic algorithm consists of three parts, each for answering one of the three questions.

C.1 Part I – Answering Question 1

For Question 1, the basic scheduling algorithm consists of two steps.

- *Step 1: Request.* If a packet arrives at I_i destined to O_j, the scheduler checks F_{ij}. If it is empty, I_i sends a request to O_j; otherwise, I_i does not send any request.
- *Step 2: Grant.* If O_j receives any requests, it chooses one to grant in a round-robin manner. That is, it will receive a binary vector representing the requests sent by the inputs. It picks the highest priority element based on its round-robin pointer and grants the corresponding input. Then it increments its round-robin pointer by one beyond the granted input.

In each time slot, since there is at most one packet arriving at each input, an input needs to send a request to at most one output and will receive no more than one grant. Therefore, the input will send the packet (or let the packet cut-through) as long as it receives a grant. The entire cut-through operation can be done by a single iteration of any iterative matching algorithm.

C.2 Part II – Answering Question 2

For question 2, the basic scheduling algorithm simply connects the inputs to the receivers according to the following schedule:

- At time slot t, the packet from I_i will be sent to R_r where $r = [(i+t) \mod N] + 1$.

Note that instead of a fixed one-to-one connection, the inputs are connected to the receivers in a round-robin fashion for better load balancing. As an example, according to our simulation, in an 8×8 OpCut switch, when there is a fully-loaded input port, the maximum overall throughput is around 0.85 if the inputs are connected to the receivers in the above way, versus 0.70 with fixed connection.

C.3 Part III – Answering Question 3

For Question 3, the scheduler requires one decision making unit for each output and one decision making unit for each buffer. It then runs the well-known iSLIP algorithm [16] between the receivers and the outputs. Each iteration of the algorithm consists of three steps:

- *Step 1: Request.* Each unmatched output sends a request to every buffer that stores a head-of-flow packet destined to this output.
- *Step 2: Grant.* If a buffer receives any requests, it chooses one to grant in a round-robin manner. That is, it will receive a binary vector representing the requests sent by the outputs. It picks the highest priority element based on its round-robin pointer and grants the corresponding output. The pointer is incremented to one location beyond the granted, if and only if the grant is accepted in Step 3.
- *Step 3: Accept.* If an output receives any grants, it chooses one to accept in a round-robin manner. That is, it will receive a binary vector representing the grants sent by the buffers. It picks the highest priority element based on its round-robin pointer and accepts the grant from the corresponding buffer. Then it increments its round-robin pointer by one beyond the granted buffer.

At the end of the algorithm, the scheduler informs each buffer which packet to transmit. It does so by sending the portion of the packet's timestamp that is needed for the buffer to locate the packet. With that information, the targeted packet can be found in constant time and sent through the transmitter. The switch is configured accordingly to route the packets to their destined output port.

IV. Pipelining Packet Scheduling

Our simulation results show that the basic scheduling algorithm introduced above can achieve satisfactory average packet delay. However, in a high speed or ultra high speed environment, it may become difficult for the scheduler to compute a schedule in each single time slot. In such a case, we can pipeline the packet scheduling to relax the time constraint. Furthermore, by pipelining multiple low-complexity schedulers, we may achieve performance comparable to a scheduler with much higher complexity. In this section we present such a pipeline mechanism.

A. Background and Basic Idea

With pipelining, the computing of a schedule is distributed to multiple sub-schedulers and the computing of multiple schedules can be overlapped. Thus, the computing of a single schedule can span more than one time slot and the time constraint can be relaxed. Another consideration here is related to fairness. By adopting the iSLIP algorithm in the third step (i.e., determining the matching between electronic buffers and switch outputs), the basic scheduling algorithm ensures that no connection between buffers and outputs is starved. However, there is no such guarantee at the flow level. In addition, as mentioned earlier, a packet that resides in the switch for too long may lead to packet dropping. To address this problem and achieve better fairness, it is generally a good idea to give certain priority to "older" packets during scheduling.

Combining the above two aspects, the basic idea of our pipelining mechanism can be described as follows. We label each flow based on the oldness of its head-of-flow packet. Among all flows destined to the same output, a flow whose head-of-flow packet has the oldest timestamp is called the oldest flow of that output. Note that there may be more than one oldest flow for an output. Similarly, the flows with the i_{th} oldest head-of-flow packets are called the i_{th} oldest flows. Instead of taking all flows into consideration, we consider only up to the

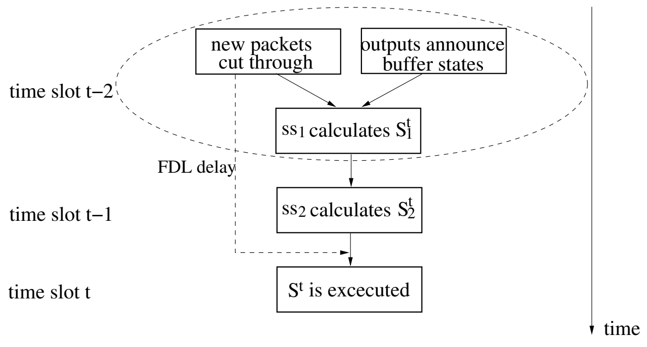


Fig. 2. Timeline of calculating schedule S^t for time slot t.

k_{th} oldest flows for each output when scheduling packets from the electronic buffer to the switch output. This may sound a little surprising but later we will see that the system can achieve good performance even when k is as small as 2. Then the procedure of determining a schedule is decomposed into k steps, with step i handling the scheduling of the i_{th} oldest flows. By employing k sub-schedulers, the k steps can be pipelined. Like the basic scheduling algorithm, the pipelined algorithm maintains packet order since only head-of-flow packets are qualified for being scheduled.

Next we will present the pipeline mechanism in more detail. Basically, like in prioritized-iSLIP [16], the flows are classified into different priorities. In our case the prioritization criterion is the oldness of a flow. By pipelining at the priority level, each sub-scheduler deals with only one priority level and does not have to be aware of the prioritization. Furthermore, since each sub-scheduler only works on a subset of all the scheduling requests, on average it converges faster than a single central scheduler. To explain how the mechanism works, we will start with the simple case of $k = 2$, that is, using only the oldest flows and second oldest flows when scheduling. We will also show that when $k = 2$, a common problem in pipelined scheduling, called duplicate scheduling, can be eliminated in our mechanism. Later we will extend the mechanism to allow an arbitrary k, and discuss potential challenges and solutions.

B. *Case of* $k = 2$

With $k = 2$, two sub-schedulers, denoted as ss_1 and ss_2 are needed to pipeline the packet scheduling. ss_1 tries to match buffers with the oldest flows to the output ports, while ss_2 deals with buffers with the second oldest flows. The timeline of calculating the schedule to be executed in time slot t, denoted as S^t, is shown in Fig. 2. The calculation takes two time slots to finish, from the beginning of time slot $t-2$ to the end of time slot $t-1$. When time slot t starts, S^t is ready and will be physically executed during this time slot. In time slot $t-2$, the cut-through operation for t is performed and the result is sent to the sub-schedulers, so that the sub-schedulers know in advance which output ports will not be occupied by cut-through packets at time t. To provide the delay necessary to realize pipelining, a fiber delay line with fixed delay of two time slots are appended to each input port. As a result, newly arrived packets are attempted for cutting-through at the beginning of time slot $t-2$, but they do not physically cut-through and take up corresponding output ports until time slot t. Later in Section IV-D we will discuss how this extra delay introduced by pipelining may be minimized. As mentioned in Section III-C, since the calculation of cutting-through is very simple and can be done by iSLIP with one iteration, or 1SLIP, there is no need to pipeline this step.

At the same time of cutting-through operation, each output port checks the buffered packets from all flows and finds its oldest and second oldest flows, as well as in which buffer these flows are stored. The outputs then announce to each buffer its state. The state of a buffer consists of two bits and has the following possible values: 0 if this buffer contains neither oldest nor second oldest flow for the output; 2 if the buffer contains one second oldest flow but no oldest flow; 1 otherwise. A buffer is said to contain an i_{th} flow of an output if it contains the head-of-flow packet of that flow. Note that the state being 1 actually includes two cases, i.e. the buffer has an oldest flow only, or has both an oldest and a second oldest flow. The point here is that we do not need to distinguish between these two cases. This is due to the fact that in a time slot at most one packet can be transmitted from a buffer to the switch output. Then if a buffer has an oldest flow for an output and a packet is scheduled from this buffer to the output, no more packets from other flows can be scheduled in the same time slot; on the other hand, if no packet from the oldest flow is scheduled to the output, no packet from the second oldest flows can be scheduled either since otherwise a packet from the oldest flow should have been scheduled instead. Thus as long as a buffer contains an oldest flow for an output, we do not need to know whether it contains a second oldest flow for that output or not.

Fig. 3 provides a simple example with $N = 3$ that shows how the announcing of oldest and second oldest flows works. In this example, we focus on one tagged output and three flows associated with it. As shown in the figure, packets p_1 and p_2 arrive in the same time slot but from different flows. p_3 arrives following p_2. A few time slots later, p_4 belonging to flow 3 arrives. We assume that some time later p_1, p_2 and p_4 become the head-of-flow packet for the three flows, respectively. It can be seen that flows 1 and 2 are the oldest flows, and flow 3 is the second oldest flow. As shown in the figure, assume that p_1 and p_2 are stored in buffers 1 and 2, respectively, and both p_3 and p_4 are in buffer 3. Then the tagged output will make the announcement as "1" to buffers 1 and 2, and "2" to buffer 3, which informs the sub-schedulers that buffers 1 and 2 have an oldest flow for this output , and buffer 3 has a second oldest flow but no oldest flow for this output.

After receiving the result of cutting-through operation, and the announcements from the outputs, sub-scheduler ss_1 is now set to work. Note that while the sub-schedulers work directly with buffers, they essentially work with flows, in particular, head-of-flow packets, since they are the only packets eligible for transmission for the sake of maintaining packet order. Denote the set of output ports that will not be occupied by cut-through packets at time slot t as O^t. What ss_1 does is to match the output ports in O^t to the buffers containing an oldest flow of these output ports. Theoretically, this process can be done by any bipartite matching algorithm. For simplicity, the iSLIP algorithm

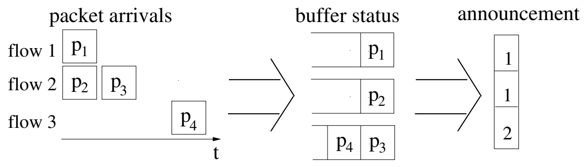


Fig. 3. An example of how an output makes the announcement. The information of all packets that are in the buffer and destined for the output port is maintained for each output port. Based on that information, an output can find the oldest and second oldest flows, and where the head-of-flow packets are buffered. Then it can make the announcement accordingly.

is adopted. In each iteration of the iSLIP algorithm, if there is more than one buffer requesting the same output port, ss_1 decides which of them the output should grant. Then, in case a buffer is granted by multiple output ports, ss_1 determines which grant the buffer should accept. The decisions are made based on the round-robin pointers maintained for each output port and buffer. The number of iterations to be executed depends on many factors, such as performance requirement, switch size, traffic intensity, etc. Nevertheless, as mentioned earlier, it can be expected that the result will converge faster than that of a single central scheduler since the sub-scheduler handles only a subset of all the scheduling requests.

ss_1 has one time slot to finish its job. At the beginning of time slot $t-1$, ss_1 sends its result to the output ports so that the output ports can update the VIQs and announce the latest buffer state. Meanwhile, ss_1 relays the result to ss_2. The functionality of ss_2 is exactly the same as ss_1, i.e. matching output ports to buffers according to some pre-chosen algorithm. The difference is that, ss_2 only works on output ports that are in O^t and are not matched by ss_1, and buffers that are announced with state 2 by at least one of these output ports. When ss_2 finishes the job at the end of time slot $t-1$, the matching based on which the switch will be configured in time slot t is ready. Meanwhile the packets that arrived at the beginning of time slot $t-2$ have gone through the two-time-slot-delay FDLs and reached the switch input. In time slot t, the buffers are notified which packet to send, and the switch is configured accordingly. Packets are then transmitted to the switch output, either directly from the switch input or from the electronic buffer.

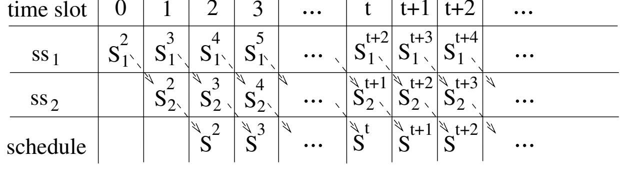


Fig. 4. The pipelined scheduling procedure for $k = 2$.

The complete picture of the pipeline packet scheduling for $k = 2$ is shown in Fig. 4. As mentioned earlier, S^t is the schedule executed in time slot t. S_i^t denotes the part of S^t that is computed by sub-scheduler ss_i during time slot $t-i$.

A potential problem with pipelined scheduling algorithms is that it is possible for a packet to be included in multiple schedules, or, being scheduled for more than once. This is called duplicate scheduling. It could occur under two different conditions: 1) in the same time slot, different schedulers may try to include the same packet to their respective schedule, since a scheduler is not aware of the progress at other schedulers in the same time slot; 2) with pipelining, there is usually a delay between a packet being included in a schedule and the schedule being physically executed. During such interval the packet may be accessed by another scheduler that works on the schedule for a different time slot. In other words, a scheduler may try to schedule a packet that was already scheduled by another scheduler but has not been physically transmitted yet.

Duplicate scheduling of a packet leads to waste of bandwidth resources, which consequently causes underutilization of bandwidth and limits throughput. In an input-queued switch, when a packet p is granted for transmission more than once by different sub-schedulers, extra grants may be used to transmit the packets behind p in the same VOQ if the VOQ is backlogged. On the other hand, if the VOQ is empty, all but one grants are wasted. With the OpCut switch architecture, the consequence of duplicate scheduling is even more serious, in that extra grants for a packet cannot be used to transmit packets behind it in the same buffer. This is due to the fact that in an OpCut switch packets from the same flow may be distributed to different buffers, and a buffer may contain packets from different flows.

Duplicate scheduling is apparently undesirable but is usually difficult to avoid in pipelined algorithms. For example, the algorithms in [17] [18] [19] all suffer from this problem, even with only two-step pipelining. It was proposed in [19] to use pre-filter and post-filter functions to reduce duplicate scheduling. However, on one hand, these functions are quite complex, and on the other hand, the problem cannot be eliminated even with those functions. The difficulty roots in the nature of pipelining, that schedulers may have to work with dated information, and the progress at one scheduler is not transparent to other schedulers. Fortunately, as will be seen next, when $k = 2$ our mechanism manages to overcome this difficulty and completely eliminates duplicate scheduling.

First of all, it is worth noting that the "oldness" of a flow is solely determined by the timestamp of its head-of-flow packet. Thus we have the following simple but important lemma.

Lemma 1: Unless its head-of-flow packet departs, a flow cannot become "younger."

Next we deal with the first condition that may lead to duplicate scheduling. That is, we show that in any time slot the two sub-schedulers will not include the same packet in their respective schedule. In fact we have a even stronger result here, as shown by the following theorem:

Theorem 1: During any time slot, sub-scheduler ss_1 and ss_2 will not consider the same flow when computing their schedule. In other words, if we denote F_i^t as the set of flows that ss_i takes into consideration in time slot t, then $F_1^t \cap F_2^t = \emptyset$ for any $t \geq 0$.

Proof: First note that for $t = 0$ there is no second oldest flow, $F_2^t = \emptyset$, thus the theorem holds. Now assume for some $t > 0$, the theorem held up to time slot $t-1$ but not in time slot t. In other words, there exists a flow f such that $f \in F_1^t$ and $f \in F_2^t$. Note that $f \in F_2^t$ indicates f was not an oldest flow at time $t-1$. Thus at $t-1$ there existed at least one flow that was older than

f and destined to the same output as f. Denote such an flow as f', then $f' \in F_1^{t-1}$ since it was an oldest flow at that time. Besides, it can be derived that no packet from f' was scheduled by ss_1 in time slot $t-1$. Otherwise, the corresponding output port should be matched, and at time t ss_2 would not consider any flow associated with that output, including f.

Furthermore, since $f' \in F_1^{t-1}$, it follows that $f' \notin F_2^{t-1}$, given that the theorem held in time slot $t-1$. Then neither ss_1 nor ss_2 could schedule any packet belonging to f' in time slot $t-1$. According to Lemma 1, f' is still older than f at time slot t. Consequently, f is not an oldest flow at t, and $f \in F_1^t$ cannot hold, which contradicts the assumption. This implies that the theorem must hold for time slot t if it held for time slot $t-1$. That proves the theorem for $t \geq 0$. ■

Next we consider condition 2. It is possible for condition 2 to occur between S_2^t and S_2^{t+1} due to the existence of a time glitch: the buffer states based on which S_2^{t+1} is calculated are announced at the beginning of time slot t. At that time S_2^t is not calculated yet. Thus it is possible that a packet is included in both S_2^t and S_2^{t+1}. In contrast, S_2^t and S_1^{t+1} can never overlap, since the latter is calculated based on the information announced after being updated with S_2^t. For the same reason, sub-schedules S_i^t and S_j^{t+x} would never include the same packet for any $t \geq 0, i, j \in \{1, 2\}$, as long as $x > 1$. Thus the task of eliminating condition 2 reduces to making sure that S_2^t and S_2^{t+1} do not overlap, which can be achieved as follows.

When an output makes its announcement, instead of three possible states as introduced earlier in this section, each buffer may be in a forth state denoted by value 3 (this is doable since the state of a buffer is 2-bit long), which means that this buffer contains a third oldest flow and no oldest or second oldest flow for this output. Furthermore, we call a flow a *solo flow* if it is the only i_{th} oldest flow, and a buffer a *solo buffer* for an output port if it contains a solo flow of that output port. Now suppose ss_2 matched an output port *op* to a buffer bf in S_2^t based on the announcements in time slot $t-2$. Then when S_2^{t+1} is being computed, bf is excluded from S_2^{t+1} if *op* again announced bf as a state-2 buffer. On one hand, if there exists at least one buffer other than bf that was announced with state 2 by *op* in time slot $t-1$, ss_2 will work with these buffers. On the other hand, if bf was a solo buffer for *op* based on the announcement at time slot $t-1$, ss_2 will work on state-3 buffers instead. Consequently, we have the following theorem.

Theorem 2: The method introduced above ensures that S_2^t and S_2^{t+1} will not introduce duplicate scheduling of a packet.

Proof: First, S_2^t and S_2^{t+1} may include the same packet only if ss_2 matches a buffer to the same output port in both S_2^t and S_2^{t+1}. Hence it is assumed that buffer bf is matched to output port *op* in both time slots $t-1$ and t by ss_2 (As a reminder, S_2^t is calculated in time slot $t-1$ based on output announcements made in time slot $t-2$). For this to occur, the state of bf announced at time slot $t-1$ can only be 3 according to the above method. Besides, bf cannot be a state-1 buffer of *op* for time slots $t-2$ and $t-1$, since otherwise bf should not be considered by ss_2. Then the states of bf announced by *op* at time slots $t-2$ and $t-1$, based on which S_2^t and S_2^{t+1} are calculated respectively, have only two possible combinations: 2 at time slot $t-2$ and 3 at time slot $t-1$ ($\{2, 3\}$), or 3 at time slot $t-2$ and 3 at time slot $t-1$ ($\{3, 3\}$). We will show that under neither of the combinations could duplicate scheduling occur.

- *{2, 3}:* In this case, by matching bf to *op*, S_2^t actually schedules to *op* the head-of-flow packet of some second oldest flow announced by *op* at time slot $t-2$. The head-of-flow packet is buffered in bf. Similarly, S_2^{t+1} schedules to *op* the head-of-flow packet of a third oldest flow announced at time slot $t-1$. Denote the two head-of-flow packets as p_a and p_b, and the two flows as f_a and f_b. On one hand, if flow f_a and flow f_b are different, packet p_a and packet p_b must be different. On the other hand, if flow f_a and flow f_b are the same flow, packet p_a and packet p_b are still different according to Lemma 1, since the flow becomes "younger" (second oldest at time slot $t-2$ and third oldest at time slot $t-1$).
- *{3, 3}:* Given that the state of bf is announced as 3 at time slot $t-1$ but ss_2 takes it into consideration when computing S_2^{t+1}, it must be true that in S_2^t ss_2 grants a buffer with a second oldest flow of *op* announced at time slot $t-2$ and that buffer is a solo buffer of *op*, which cannot be bf whose state announced at time slot $t-2$ is 3. This contradicts with the assumption that bf is matched to output port *op* in both time slots by ss_2.

Combining the two cases, the theorem is proved. ■

By now, duplicate scheduling is completely ruled out in our mechanism.

It is worth pointing out that the scheduler will not omit any packet in the buffer. First, the scheduler always maintains packet order in a flow; therefore, a packet will not get skipped within a flow and will eventually become the head-of-flow packet. Second, a flow will either has its head-of-flow packet scheduled, or as Lemma 1 indicates, will eventually become an oldest flow. Since all oldest flows are serviced in a round-robin manner, it is guaranteed that no flow will get starved if the switch is not overloaded, and any head-of-flow packet will be scheduled within one round-robin cycle.

C. *Case of* $k > 2$

We now extend our result for $k = 2$ to the case that k is an arbitrary integer between 3 and N. The system performance can be improved at the cost of extra subschedulers. While the basic idea remains the same as $k = 2$, there are a few implementation details that need to be addressed when k becomes large. Duplicate scheduling can no longer be eliminated with an arbitrary k due to the increased scheduling complexity. Nevertheless we will propose several approaches to reducing it.

The basic pipelined scheduling procedure is given in Fig. 5. An FDL of length k is attached to each input port to provide the necessary delay for computing the schedules. k identical sub-schedulers, ss_1, ss_2, ..., ss_k are employed, ss_i dealing with buffers that contain an i_{th} oldest flow of some output port. Intermediate results are passed between adjacent sub-schedulers and used to update the VIQ status. The computing of the schedule to be executed in time slot t spans k time slots, from the beginning of time slot $t-k$ to the end of time slot $t-1$. The announcement of buffer states from an output port to the sub-schedulers can be done exactly the same way as that for $k = 2$, except that the state of a buffer for an output is now of length

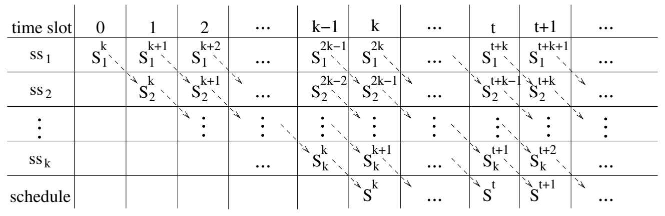


Fig. 5. Pipelined scheduling procedure for an arbitrary k.

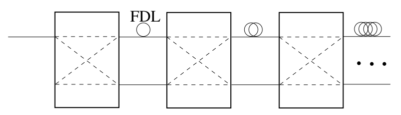


Fig. 6. A possible implementation of an FDL that can provide flexible delays to fit the needs of pipeline with different number of sub-schedulers. There are $\lfloor \log K \rfloor + 1$ stages. The i_{th} stage is able to provide either zero delay or 2^i time slot delay.

$\log(k+1)$ bits.

We have addressed the solo buffer problem for $k = 2$ to eliminate duplicate scheduling. Namely, if sub-scheduler ss_2 matched a buffer bf to an output port op in S_2^t, it will not consider bf as a state-2 buffer for op when computing S_2^{t+1} even if it was announced so. In case bf is the solo buffer of op, i.e. the buffer announced by op to contain the only second oldest flow of it, ss_2 will work on state-3 buffers for op trying to keep work conserving. For an arbitrary k, the rule is still kept, that if ss_i matched a buffer bf to an output port op in S_i^t, it will not consider bf as a state-i buffer for op when computing S_i^{t+1}. However, if bf is the solo buffer of op, ss_i will *not* turn to buffers with state $i+1$. The reason is that, while this method involves only ss_2 when $k = 2$, it may cause a chain effect when $k > 2$: if ss_i sets to work on buffers with state $i+1$ at some time, then ss_{i+1} needs to work on buffers with state $i+2$ for the same schedule. In case there is only a solo buffer with state $i+1$ and is matched by ss_i again, then in the next time slot, ss_i may have to work on buffers with state $i+2$ and ss_{i+1} has to work on buffers with state $i+3$. The process could go on and become too complicated to implement. Therefore, if an output announced the same buffer as the solo buffer in two consecutive time slots, say, $t-1$ and t, and ss_i matched this buffer to the output in S_i^{t+k-i}, it will not try to match the output to any buffer in $S_i^{t+k+1-i}$. In other words, we will let ss_i be *idle* for the output in time slot $t+i$ in that case.

By allowing a sub-scheduler to be idle for some output port in certain time slot, we prevent the possibility that the sub-scheduler schedules a packet that was already scheduled and blocks other sub-schedulers behind it in the pipeline from scheduling a packet to that output port. Unfortunately, the cost is that Theorem 1 does not hold for $k > 2$. To see this, first note that F_i^t is essentially the set of the i_{th} oldest flows of every output port at the beginning of time slot $t+1-i$. For instance, F_1^t is the set of the oldest flows at time slot t and F_3^t is the set of the third oldest flows at time slot $t-2$. If there is a flow f such that $f \in F_i^t$, then it is one of the i_{th} oldest flows for some output port at time slot $t+1-i$. During the time interval, denoted as T, from time slot $t+1-i$ to time slot $t-j$ for some $j < i$, at most $i-j$ flows for that output can be scheduled. Therefore, at time slot $t+1-j$, f is at least the $i-(i-j) = j_{th}$ oldest flow. If f is indeed the j_{th} oldest flow, which can occur if and only if $i-j$ flows that are "younger" than f have been scheduled during T and all of them are solo flows, $f \in F_j^t$ holds. In that case, $f \in F_i^t \cap F_j^t$ holds, and ss_i and ss_j may schedule the same packet during time slot t. Nevertheless, as can be seen, the possibility that F_i^t overlaps with F_j^t is rather small and should not significantly affect the overall system performance. In fact, if we let P_r denote the probability that an output port op announces a buffer bf as the buffer which contains the solo second oldest flow and bf is later matched to op by ss_2 based on the announcement, then according to our simulations for $k = 4$, when the traffic intensity is as high as 0.9, P_r is less than 2%. The probability for the case of multiple solo flows is roughly exponential to P_r and thus is even smaller.

D. Adaptive Pipelining

We have discussed the mechanism to pipeline packet scheduling in the OpCut switch for any fixed k. In the following we will enhance it by adding adaptivity. The motivation is that, in our mechanism, the extra delay introduced by pipelining is equal to the number of active sub-schedulers, or k. When traffic is light, a small number of sub-schedulers may be sufficient to achieve satisfactory performance, or pipeline is not necessary at all. In this case, it is desirable to keep k as small as possible to minimize the extra delay . On the other hand, when the traffic becomes heavy, more sub-schedulers are activated. Although the delay of pipelining increases, now more packets can be scheduled to the switch output since more packets are taken into consideration for scheduling due to the additional sub-schedulers.

The first step towards making the pipelined mechanism adaptive is to introduce flexibility to the FDLs attached to the switch output ports. Since k sub-schedulers working in pipeline require a k time slot delay of the newly arrived packets, the FDL needs to be able to provide integral delays between 0 and K time slots, where K is the maximum number of sub-schedulers that can be activated. Clearly, $K \leq N$.

A possible implementation of such an FDL is shown in Fig. 6. The implementation adopts the logarithmic FDL structure [22] and consists of $\lfloor \log K \rfloor + 1$ stages. A packet encounters no delay or 2^i time slot delay in stage i, depending on the input port it arrives at the switch of stage i and the state of the switch. Through different configurations of the switches, any integral delay between 0 and K can be provided.

The number of packet arrivals in each time slot is recorded, and the average over recent W time slots is calculated and serves as the estimator of current traffic intensity. This average value can be efficiently calculated in a sliding window fashion: let A_i denote the number of packet arrivals in time slot i, then at the end of time slot t, A is updated according to $A = A - (A_{t-w+1} - A_t)/W$. An arbitrator decides whether a sub-scheduler needs to be turned on or off based on A. If during certain consecutive time slots, A remains larger than a preset threshold for the current value of k, an additional sub-scheduler will be put into use. Similarly, if A drops below some

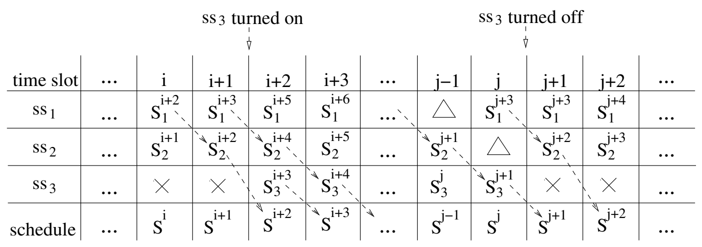


Fig. 7. An example of sub-schedulers being turned on and off.

threshold and does not bounce back in certain time interval, an active sub-scheduler can be turned off.

The value of W can be adjusted to trade-off between sensitivity and reliability: if W is large, the averaging of traffic intensity is over a relatively long time period, and it is less likely that a small jitter will trigger the activation of an additional sub-scheduler. However, more time is needed for it to detect a substantial increase in traffic intensity, and vice versa.

An example of adaptive pipelining is given in Fig. 7. The basic idea is the same for any k value, thus we only show the process from two sub-schedulers to three sub-schedulers and then back to two to keep it neat. The "$\times$" state in the figure indicates the sub-scheduler is off, and a "Δ" means the sub-scheduler is on but will be idle in the time slot. The arrows in the figure illustrate how the intermediate results are relayed among sub-schedulers at transition points when a sub-scheduler is being turned on or off.

V. Performance Evaluation

In this section, we evaluate the performance of the switch under two widely used traffic models: the uniform Bernoulli traffic and the non-uniform bursty traffic. Both models assume that the arrival at an input port is independent of other input ports. The uniform Bernoulli traffic assumes that the packet arrival at an input port is a Bernoulli process and the destination of an arrived packet is uniformly distributed over all output ports. The non-uniform bursty traffic assumes that an input port alternates between the "on" state and the "off" state, with the length of a state following geometric distribution. If and only if in an "on" state, a packet arrives at the input port at the beginning of every time slot. Therefore traffic intensity is given by $l_{on}/(l_{on}+l_{off})$, where l_{on} and l_{off} denote the average length of on and off states in terms of time slots, respectively. All packets arriving during the same "on" period are destined for the same output port and form a burst. Same as in [2], l_{on} (or, equivalently, the average burst length) is set to 10 in our simulations. A packet arrived at I_i is destined to O_i with probability $\mu + (1-\mu)/N$ and is destined to O_j with probability $(1-\mu)/N$ for $j \neq i$, where μ is the "unbalance factor" and is set to be 0.5 which is the value that results in the worst performance according to [20]. We have evaluated OpCut switches of different sizes with both non-pipelined and pipelined schedulers. Each simulation was run for 10^6 time slots.

We implemented two instances of the proposed pipelining mechanism, denoted as p-k2-2SLIP and p-k4-2SLIP, respectively. Both of them are built on sub-schedulers executing two steps of iSLIP in each time slot. p-k2-2SLIP runs two such sub-schedulers and covers up to the second oldest flows of each input port, while p-k4-2SLIP runs four sub-schedulers and covers up to the fourth oldest flows. For comparison purpose, we implemented the basic non-pipelined scheduler running iSLIP as well, denoted as np-iSLIP. Also included in the simulations are the straightforwardly pipelined iSLIP scheduler (denoted as p-iSLIP) - i sub-schedulers, each of which executes one iteration of iSLIP in a time slot. Unlike the proposed pipelined schedulers, the straightforward approach is not aware of the duplicate scheduling problem.

A. Cut-Through Ratio

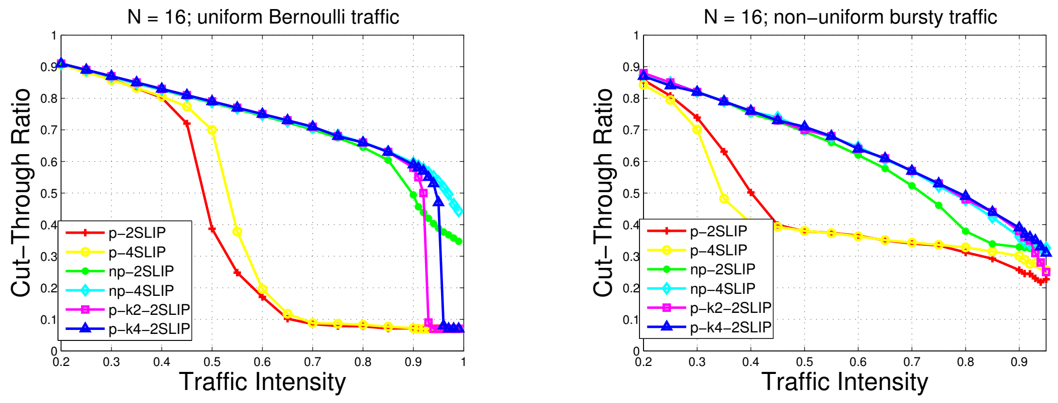
First we investigate the packet cut-through ratio, which indicates how much portion of packets can cut-through the switch without experiencing electronic buffering. Apparently, if only a tiny portion of packets could cut-through, or packets could cut-through only when the traffic intensity is light, the OpCut switch would not be very promising. From Fig. 8, we can see that when the load is light, the cut-through ratio is high with all schedulers under both traffic models and switch sizes. However, For p-iSLIP schedulers, the ratio drops sharply with the increment in traffic intensity. For all the other simulated schedulers, the ratio decreases much slower, and stays above 60% under Bernoulli uniform traffic and 30% under bursty non-uniform traffic even when the load rises to 0.9.

We notice that under Bernoulli uniform traffic, there is a sharp drop in the cut-through ratio for both pipelined schedulers. For the 16×16 switch, the drop occurs at 0.93 load for p-k2-2SLIP and 0.95 load for p-k4-2SLIP. For the 64×64 switch it occurs at slightly higher loads. As will be confirmed shortly by the average packet delay, these are the points at which the OpCut switch is saturated with the respective pipelined scheduler. However, it is worth pointing out that higher cut-through ratio does not necessarily imply better overall performance. In particular, throughput is not directly related to cut-through ratio, as packets can always be transmitted from the buffers to the switch output. Thus while non-pipelined scheduler np-2SLIP results in a higher cut-through ratio than p-k2-2SLIP and p-k4-2SLIP under higher-than-0.93 uniform Bernoulli traffic, it can be seen from Fig 9 that the pipelined schedulers actually achieve better delay and higher throughput than np-2SLIP under that traffic model.

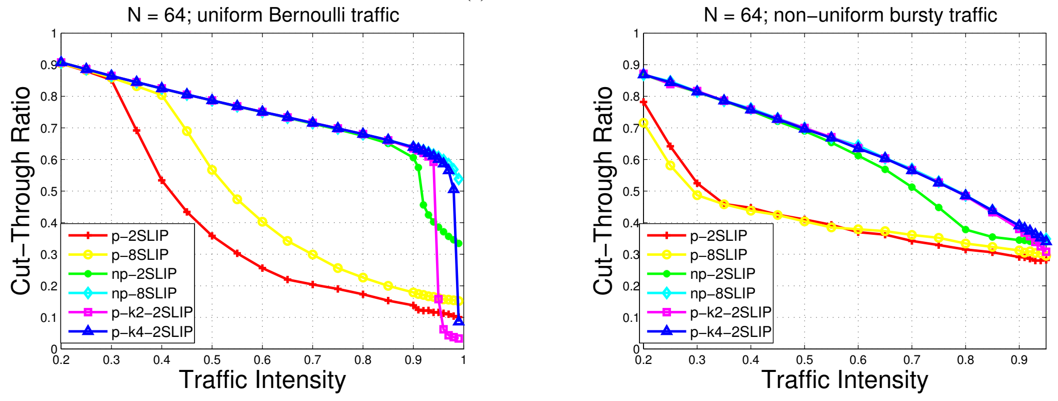
An interesting observation is that for bursty non-uniform traffic, such sharp drops in cut-through ratio do not exist. This is likely due to the nature of non-uniform traffic that, except for those hotspot flows, most flows contains much fewer packets. For those packets, cutting-through becomes easier compared to the uniform-traffic scenario, since whether a packet can cut-through is independent of packets from other flows.

B. Average Packet Delay

Fig. 9 shows the average packet delay of the OpCut switch under different schedulers, traffic models and switch sizes. The ideal output-queued (OQ) switch is implemented to provide the lower bound on average packet delay. It can be seen that the straightforward pipelining approach, p-iSLIP, performs very poor due to underutilization of bandwidth caused by the du-



(a) 16×16 switch



(b) 64×64 switch

Fig. 8. Packet cut-through ratio with non-pipelined and pipelined schedulers under different traffic models and switch sizes. p-iSLIP: pipelined iSLIP with i sub-schedulers, each executing 1SLIP. np-iSLIP: non-pipelined scheduler executing iSLIP in each time slot. p-ki-2SLIP: pipelined scheduling that takes up to the i_{th} oldest flows into consideration, each sub-scheduler executing 2SLIP.

plicate scheduling problem. On the other hand, as instances of the proposed pipelining mechanism, p-k2-2SLIP and p-k4-2SLIP lead to substantially improved performance. The maximum throughput p-k2-2SLIP can sustain is about 0.93 under uniform Bernoulli traffic and 0.9 under non-uniform bursty traffic, which outperforms np-2SLIP by about 5% and 15%, respectively. In other words, using the same 2-SLIP scheduler, the system throughput can be improved through pipelining.

In fact, except for under light uniform Bernoulli traffic where the extra delay introduced by pipelining is comparatively significant (for which we have proposed the adaptive pipelining scheme), the performance of p-k4-2SLIP is very close to that of np-4SLIP when $N = 8$ and np-8SLIP when $N = 64$, which is in turn very close to that of the OQ switch in terms of average packet delay. That is, the performance of a non-pipelined scheduler that executes 8 iterations of iSLIP in each time slot can be well emulated by four schedulers working in pipeline, each of which executes 2-iteration iSLIP only. The time constraint on computing a schedule is relaxed by four times and the system performance is hardly affected, which illustrates the effectiveness of the proposed pipelining mechanism in reducing scheduler complexity.

C. *Adaptive Pipelining*

Next we examine the effectiveness of adaptive pipelining. To illustrate the point, we consider a simple synthetic traffic model given in Fig. 10(a). The performance of adaptive pipelining is obtained and compared with that of non-pipelining and pipelining with a fixed number of sub-schedulers. All schedulers, pipelined or not, are assumed to run 1SLIP. The average packet delay is sampled every 100 time slots and is plotted against time in Fig. 10(b). It can be seen that in the first 2×10^4 time slots, while non-pipelining and 2-subscheduler pipelining eventually lead to very large delay, 3-subscheduler pipelining and adaptive pipelining successfully sustain the increase in traffic intensity. Moreover, when traffic intensity drops back to around 0.5, adaptive pipelining outperforms 3-subscheduler pipelining as it does not have a fixed 3 time-slot pipelining delay. In fact, according to Fig. 10(b), adaptive pipelining always has the minimum delay among all of these schedulers regardless of the change in traffic load. It achieves both high throughput under heavy traffic and low pipelining delay under light traffic by adjusting the number of sub-schedulers according to the traffic load.

VI. Related Work and Comparison

Building optical packet switches has attracted many interests in recent years, see, for example, [5], [23], [24], [25]. In this section we compare the OpCut switch with some recently pro-

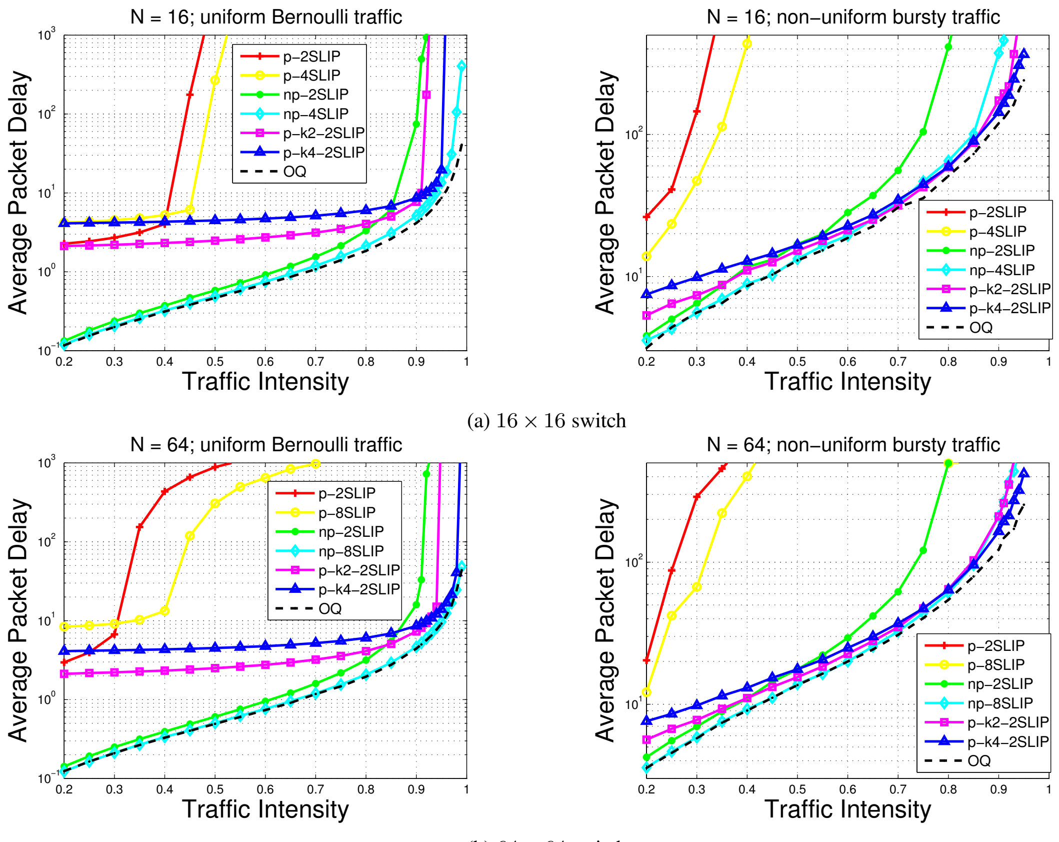


Fig. 9. Packet delay with non-pipelined and pipelined schedulers under different traffic models and switch sizes. The notations of schedulers are the same as in Fig. 8. OQ: ideal output-queued switch.

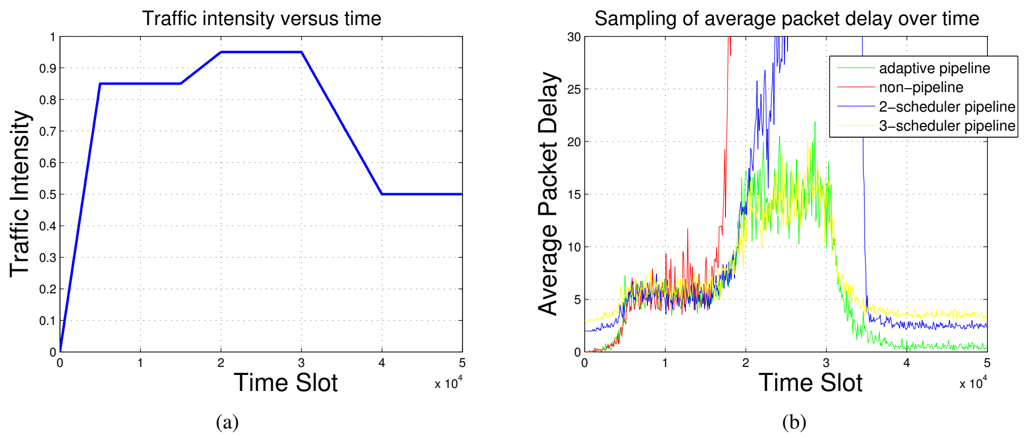


Fig. 10. An example of adaptive pipeline. (a) The traffic model under which the traffic intensity changes with time. (b) Average packet delay over time under different pipelining strategies.

posed architectures.

Earlier designs mainly focused on all-optical switches with switching and buffering both in the optical domain, such as [25]. The performance of such designs are usually constrained by the physical limit of fiber delay lines.

Recently, IBM has built some prototypes of optical interconnects with tera bit switching capacity [2], [5]. The IBM PERCS project [14] adopts optical circuit switching on fixed routes to transfer long-lived bulk data packets. If data at a capacity of a fraction of a wavelength's granularity is to be carried, some fiber capacity will be wasted. To overcome this problem, the PERCS project uses electronic packet switching for short-lived data exchanges. Thus, in this sense, it may be considered as an intermediate feasible solution before more efficient optical packet switching becomes reality.

Similar to the OpCut switch, the prototype switch of the IBM OSMOSIS project [2], [5] also uses electronic buffer to overcome the buffering problem in optical packet switching. The OSMOSIS switch adopts an input-buffered architecture. At the input ports, all packets are converted from optical to electrical to be stored in electronic memory. The packets are then converted back to optical before being switched. As a result, all packets experience the Optical-Electronic-Optical (OEO) conversion delay. On the other hand, as the simulation results show, in the OpCut switch a large percentage of the packets cut-through the switch and do not experience such delay.

The idea of recirculating buffer can be traced back to the the Starlite switch [29] and the Sunshine switch [30]. Both of them are electronic switches based on banyan networks, and rely on the combination of a Batcher sorting network, a trap network and a concentrator to achieve internal nonblocking. Their drawback is that the size of the sorting network and the concentrator needs to be very large for the switch to achieve low packet loss when heavily loaded. In [26], [27], optical switches with "recirculating buffer" were proposed and analyzed. These switches also allow the packets that cannot be sent to the output port to be sent to a buffer inside the switch. However, the buffer is optical and its size is limited by the size of the switching fabric. More importantly, the problem of maintaining packet order was not addressed in [26], [27] and packets may be out of order after exiting the switch.

The two-stage switch in [28] bears some interesting similarities to the OpCut switch. In a two-stage switch with N inputs and N outputs, there are two $N \times N$ switches with buffers sandwiched in between. The two switches follow a fixed connection pattern to connect the inputs to the outputs: at time slot t, input i is connected to output $(i+t) \mod N$. Basically, the first stage switch spreads the input packets evenly on the buffers and the second stage switch sends the packets out in a round-robin manner. Like in the OpCut switch, maintaining packet order is challenging in the two-stage switch. To solve this problem, the two-stage switch needs additional queueing management such as the three-dimensional queue proposed in [13], or a sophisticated mechanism to feedback packet departure times to the inputs [21]. A potential problem with the two-stage switch is that it results in long packet delays because a packet may have to wait for N time slots before being sent out of the switch, even when the traffic load is very light. On the other hand, as we have seen, the OpCut switch maintains packet order by simply scheduling head-of-flow packets only and achieves very low latency by allowing packets to cut-through the switch. Furthermore, the OpCut switch can adopt pipelined scheduling adaptively to achieve the best balance between performance and scheduler complexity.

VII. Conclusions

In this paper, we have considered pipelining the packet scheduling in the *OpCut* switch. The key feature of the OpCut switch is that it allows packets to cut-through the switch whenever possible, such that packets experience minimum delay. Packets that cannot cut-through are received by receivers and stored in the electronic buffer, and can be sent to the output ports by the transmitters. We have presented a basic packet scheduler for the OpCut switch that is simple to implement and achieves satisfactory performance. We then proposed a mechanism to pipeline packet scheduling in the OpCut switch by employing k sub-schedulers. The i_{th} sub-scheduler handles the scheduling of the i_{th} oldest flows of the output ports. We have respectively discussed the implementation details for $k = 2$ and an arbitrary k. For the case of $k = 2$, we have shown that our mechanism eliminates the duplicate scheduling problem. With arbitrary k, duplicate scheduling can no longer be eliminated, but we have proposed approaches to reducing it. We have further proposed an adaptive pipelining scheme to minimize the extra delay introduced by pipelining. Our simulation results show that the OpCut switch with the proposed scheduling algorithms achieve close performance to the ideal output-queued (OQ) switch in terms of packet latency, and that the pipelined mechanism is effective in reducing scheduler complexity and further improving switch throughput.

References

[1] Z. Zhang and Y. Yang, "Performance analysis of optical packet switches enhanced with electronic buffering," *Proc. of the 23th IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[2] I. Iliadis and C. Minkenberg, "Performance of a speculative transmission scheme for scheduling-latency reduction," *IEEE/ACM Trans. Networking*, vol. 16, no. 1, pp. 182-195, Feb. 2008.

[3] R.R. Grzybowski, B.R. Hemenway, M. Sauer, C. Minkenberg, F. Abel, P. Muller and R. Luijten "The OSMOSIS optical packet switch for supercomputers: Enabling technologies and measured performance," *Proc. Photonics in Switching 2007*, pp. 21-22, Aug. 2007.

[4] C. Minkenberg, et. al, "Designing a crossbar scheduler for HPC applications," *IEEE Micro*, vol. 26, pp. 58-71, May 2006.

[5] R. Hemenway, R.R. Grzybowski, C. Minkenberg and R. Luijten, "Optical-packet-switched interconnect for supercomputer applications," *Journal of Optical Networking*, vol. 3, no. 12, pp. 900-913, Dec. 2004.

[6] J.-K.K. Rhee, C.-K. Lee, J.-H. Kim, Y.-H. Won, J. S. Choi and J. Y. Choi, "Power and cost reduction by hybrid optical packet switching with shared memory buffering," *IEEE Communications Magazine*, vol. 49, pp. 102-110, May 2011.

[7] F. Xia, L. Sekaric and Y. Vlasov, "Ultracompact optical buffers on a silicon chip,", *Nature Photonics* **1**, 2007.

[8] Y. Okawachi, M.S. Bigelow, J.E. Sharping, Z. Zhu, A. Schweinsberg, D.J. Gauthier, R.W. Boyd and A.L. Gaeta, "Tunable all-optical delays via Brillouin slow light in an optical fiber," *Phys. Rev. Lett.*, **94**, 153902, 2005.

[9] T. Zhang, K. Lu and J.P. Jue, "Shared fiber delay line buffers in asynchronous optical packet switches," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 4, pp. 118 - 127, April 2006.

[10] C.-S. Chang, Y.-T. Chen and D.-S. Lee, "Constructions of optical FIFO queues," *IEEE/ACM Trans. Networking*, vol. 14, pp. 2838-2843, 2006.

[11] A.D. Sarwate and V. Anantharam, "Exact emulation of a priority queue with a switch and delay lines," *Queueing Systems: Theory and Applications*, vol. 53, pp. 115-125, Jul. 2006.

[12] N. McKeown, A. Mekkittikul, V. Anantharam and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Trans. Communications*, vol. 47, no. 8, pp. 1260-1267, Aug. 1999.

[13] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," *IEEE INFOCOM '02*, New York, June 2002.

[14] K.J. Barker, A. Benner, R. Hoare, A. Hoisie, A.K. Jones, D.J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel and P. Walker, "On the feasibility of optical circuit switching for high performance computing systems," *Proc. ACM/IEEE Conference on Supercomputing (SC '05)*, Seattle, WA, Nov. 2005.

[15] T. Anderson, S. Owicki, J. Saxe and C. Thacker, "High speed switch scheduling for local area networks," *ACM Trans. Computer Systems*, pp. 319-352, Nov. 1993.

[16] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Networking*, vol. 7, no. 2, pp. 188-201, 1999.

[17] A. Smiljanic, R. Fan and G. Ramamurthy, "RRGS-round-robin greedy scheduling for electronic/optical terabitswitches," *GLOBECOM 1999*, pp. 1244-1250, 1999.

[18] E. Oki, R. Rojas-Cessa and H. Chao, "A pipeline-based approach for maximal-sized matching scheduling in input-buffered switches," *IEEE Communication Letters*, vol. 5, pp. 263-265, Jun. 2001.

[19] C. Minkenberg, I. Iliadis and F. Abel, "Low-latency pipelined crossbar arbitration," *Proc. IEEE GLOBECOM 2004*, vol. 2, pp. 1174-1179, 2004.

[20] R. Rojas-Cessa, E. Oki, Z. Jing and H. Chao, "CIXB-1: Combined input-one-cell-crosspoint buffered switch," In Proc. *2001 IEEE Workshop on High-Performance Switching and Routing (HPSR 2001)*, pp. 324-329, Dallas, TX, May 2001.

[21] C.-S. Chang, D.-S. Lee, Y.-J. Shih and C.-L Yu, “Mailbox switch: a scalable two-stage switch architecture for conflict resolution of ordered packets,”*IEEE Trans. Communications*, vol. 56, no. 1, pp. 136-149, Jan. 2008.

[22] D.K. Hunter, D. Cotter, R.B. Ahmad, W.D. Cornwell, T.H. Gilfedder, P.J. Legg, and I. Andonovic, “Buffered switch fabrics for traffic routing, merging, and shaping in photonic cell networks,” *Journal of Lightwave Technology*, vol. 15, pp. 86-101, Jan. 1997.

[23] B.E. Lemoff, et. al., “Demonstration of a compact low-power 250-Gb/s parallel-WDM optical interconnect,” *IEEE Photonics Technology Letters*, vol. 17, no. 1, pp. 220-222, Jan. 2005.

[24] J. Gripp, M. Duelk, J.E. Simsarian, A. Bhardwaj, P. Bernasconi, O. Laznicka and M. Zirngibl, “Optical switch fabrics for ultra-high-capacity IP routers,” *J. Lightwave Technology*, vol. 21, no. 11, pp. 2839-2850, Nov. 2003.

[25] S.L. Danielsen, C. Joergensen, B. Mikkelsen and K.E. Stubkjaer, “Analysis of a WDM packet switch with improved performance under bursty traffic conditions due to tunable wavelength converters,” *J. Lightwave Technology*, vol. 16, no. 5, pp. 729-735, May 1998.

[26] C. Develder, M. Pickavet and P. Demeester, “Assessment of packet loss for an optical packet router with recirculating buffer,” *Optical Network Design and Modeling (ONDM) 2002*, pp. 247-261, Torino, Italy, 2002.

[27] Z. Zhang and Y. Yang, “WDM optical interconnects with recirculating buffering and limited range wavelength conversion,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 5, pp. 466-480, May 2006.

[28] C.S. Chang, D.S. Lee, and Y.S. Jou, “Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering,” *Computer Communications*, vol. 25, pp. 611-622, Apr. 2002.

[29] A. Huang and S. Knauer, “Starlite: A wideband digital switch,” *Proc. GOLBECOM ’84*, Nov. 1984.

[30] J.N. Giacopelli, J.J Hickey, W.S. Marcus, W.D. Sincoskie and M. Littlewood, “Sunshine: A high-performance self-routing broadband packet switch architecture,” *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1289-1298, Oct. 1991.