Pipelining Packet Scheduling in a Low Latency Optical Packet Switch

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 switching architectures with electronic buffers have been proposed to tackle the lack of optical Random Access Memories (RAM). 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. In an OpCut switch, a packet is converted and sent to the electronic buffers only if it cannot be directly routed to the switch output. As the length of a time slot shrinks with the increase of the line card rate in such a high-speed system, it may become too stringent to calculate a schedule in each single time slot. In such a case, pipelining scheduling can be adopted to relax the time constraint. In this paper, we present a novel mechanism to pipeline the packet scheduling in the OpCut switch by adopting multiple "sub-schedulers." The computation of a complete schedule for each time slot is done under the collaboration of sub-schedulers and spans multiple time slots, while at any time schedules for different time slots are being calculated simultaneously. We present the implementation details when two sub-schedulers are adopted, and show that in this case our pipelining mechanism eliminates duplicate scheduling which is a common problem in a pipelined environment. With an arbitrary number of sub-schedulers, the duplicate scheduling problem becomes very difficult to eliminate due to the increased scheduling complexity. Nevertheless, we propose several approaches to reducing it. Finally, to minimize the extra delay introduced by pipelining as well as the overall average packet delay under all traffic intensities, we further propose an adaptive pipelining scheme. Our simulation results show that the pipelining mechanism effectively reduces scheduler complexity while maintaining good system performance.

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

I. INTRODUCTION

In recent years, switching networks draw increasingly more attentions 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 a switching fabric, 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 the switching fabric. Some optical switch prototypes have been built and exhibited, for example, the recent PERCS (Productive, Easy-to-use, Reliable Computing System) project [13] and OSMOSIS (Optical Shared Memory Supercomputer Interconnect System) project [3][4][5] at IBM. It has gradually become consensus that future high speed switches should exploit as many advantages optics can provide as possible.

One of the major problems with current optical technology 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 [8] [9] [10], but the random accessibility is still absent.

Alternatively, there are emerging techniques that aim at slowing the light down, for example, [6][7]. 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.

A low latency optical switching architecture that combines optical switching with 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 OpCut switch is equipped with recirculating electronic buffers, but an optical packet that arrives at the switch input is converted into electronics and sent to the electronic buffers if and only if it cannot be directly routed to the switch output. Packets that are electronically buffered can be converted back to optics and sent to the switch output later. By allowing packets to "cut-through" the switch, the packets experience low latency, and the number of O/E/O conversions is minimized. A core component of this switch is the packet scheduler, which takes care of scheduling of packets from the input directly to the output, from the input to the electronic buffers, and from the buffers to the output.

There has been a lot research on scheduling 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 match-*

Research supported by US National Science Foundation under grant numbers CCF-0915823 and CCF-0915495.

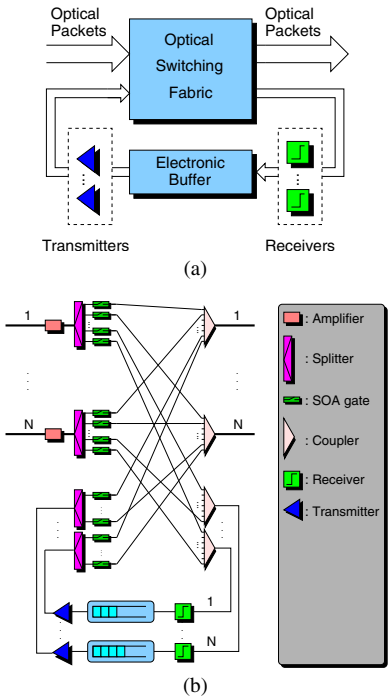


Fig. 1. (a) A high level view of the OpCut switch. (b) A possible implementation of the OpCut switch.

ing 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) [11]. 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) [16], Parallel Iterative Matching (PIM) [14] and iSLIP [15]. 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 can be adopted to relax the time constraint. In [16] 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 [17] 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 [18], PMM is "more a parallelization than a pipeline" since there is no information exchange between schedulers. [18] 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 a 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 provide a mechanism to pipeline the packet scheduling in the OpCut switch. The rest of this paper is organized as follows. Section II briefly introduces the OpCut switch architecture and the basic packet scheduler in this switch. Section III presents the pipelining mechanism for packet scheduling in the OpCut switch. Section IV describes an adaptive pipelined packet scheduling scheme. Section V presents the simulation results. Finally, Section VI concludes the paper.

II. The OpCut Switch

In this section we introduce the OpCut switch [1]. We will give a brief description of the switch architecture, followed by the discussion on its basic packet scheduler.

A. Switch Architecture

The OpCut switch works in time slots. Packets are of fixed length and 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]. In this paper, we follow the same assumptions as other optical switch designs that transmissions are synchronized and packets arrive at the switch at the beginning of time slots. In 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. Whenever possible, arrived 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. 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.

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). There are N receivers, which equals the number of input/output ports. The N receivers are necessary to avoid packet loss, since in the worst case there may be N packet arrivals in a single time slot, and none of them can cut-through the switch. 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 electronic buffers maintain no queues. Instead, packets are indexed by their timestamp. Since in each time slot a receiver picks up at most one packet, it is possible to locate a packet in a buffer in constant time given the timestamp of the packet. As an implementation detail, the buffer can be used in a wrap-around fashion thus does not need to have very large capacity. For instance, if the index is 8-bit long, then each buffer stores up to 256 packets. Consequently, only the lower 8 bits

of the timestamp are 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.

B. Packet Scheduling in the OpCut Switch

The basic scheduler for the OpCut switch breaks a schedule into three parts: a matching between switch input ports and output ports for cutting-through the newly arrived packets, a matching between switch inputs and the receivers for picking up the packets that cannot cut-through, and a matching between the electronic buffers and the switch output ports for the transmission of buffered packets. Here we adopt a simple strategy to maintain packet order, which is generally desired for switches [20], [12]. Basically, the scheduler allows a packet to be sent to an output only if this packet is a head-of-flow packet. That is, if the packet has the oldest timestamp among all packets from the same flow that are currently waiting for transmission to the switch output. In each time slot, the basic scheduler sends the maximum number of packets to the switch output without violating packet order.

At the beginning of a scheduling cycle, cut-through operation is first executed. If an input port has a newly arrived packet, it may send a scheduling request to the desired output port. To ensure packet order, such a request can be sent if and only if no packet belonging to the same flow as the newly arrived packet is now being stored in the electronic buffers. Since at most one packet can arrive at an input port during one time slot, each input port sends at most one request. Hence, it takes only one iteration of an iterative matching algorithm, such as iSLIP, to handle all those requests.

If there are packets that fail to cut-through, they will be picked up by receivers according to the following rule:

$$r = [(i + t) \mod N] + 1 \tag{1}$$

where r is the receiver index, i is the input index of the packet, t is current time slot, and N is the switch size. 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 overall throughput is around 0.85 if the inputs are round-robinly connected to the receivers, versus 0.70 with fixed connection.

At the same time, the scheduling is computed for the packets from the buffers to the output ports that are not occupied by cut-through packets. Again, to maintain packet order, only the head-of-flow packets are eligible for being scheduled. 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 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. With the VIQs, the scheduler can adopt any bipartite matching algorithm, for example, the iSLIP algorithm, to determine the matching between the electronic buffers to the switch output.

III. 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. 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 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 describe the pipeline mechanism in more detail. Basically, like in prioritized-iSLIP [15], 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,

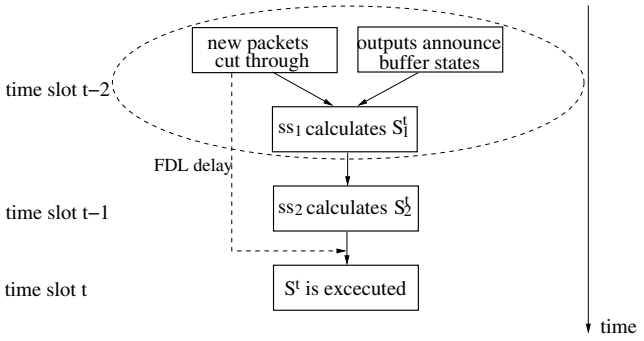


Fig. 2. Timeline of calculating schedule S^t for time slot t.

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 we will discuss how this extra delay introduced by pipelining may be minimized. As mentioned in Section II-B, 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.

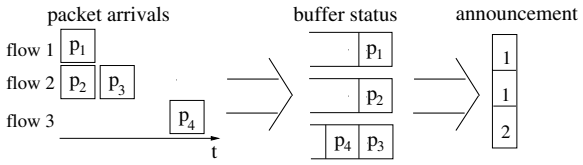


Fig. 3. An example of how an output makes the announcement. The information of all packets that are in the buffer and destined for the output port is maintained for each output port. Based on that information, an output can find the oldest and second oldest flows, and where the head-of-flow packets are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

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 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.

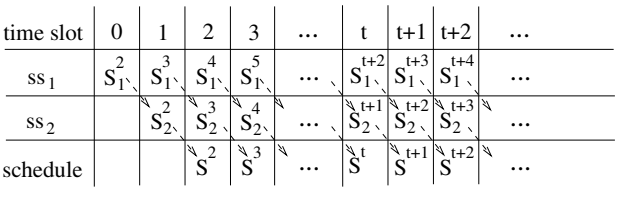


Fig. 4. The pipelined scheduling procedure for $k = 2$.

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 an 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, as the 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 [16] [17] [18] all suffer from this problem, even with only two-step pipelining. It was proposed in [18] 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 an 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 that 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 the 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 (Recall that 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.

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.

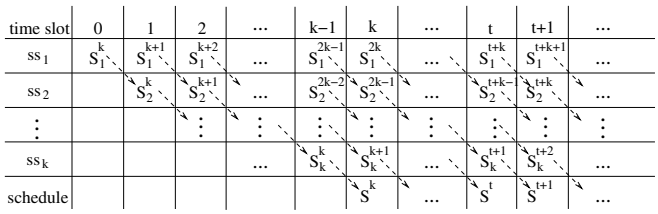


Fig. 5. Pipelined scheduling procedure for an arbitrary k.

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 $\log(k+1)$ 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 contains 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 a 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.

IV. 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 [21] 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 fash-

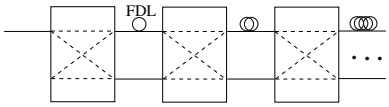


Fig. 6. A possible implementation of an FDL that can provide flexible delays to fit the needs of pipeline with different number of sub-schedulers. There are $\lfloor \log K \rfloor + 1$ stages. The i_{th} stage is able to provide either zero delay or 2^i time slot delay.

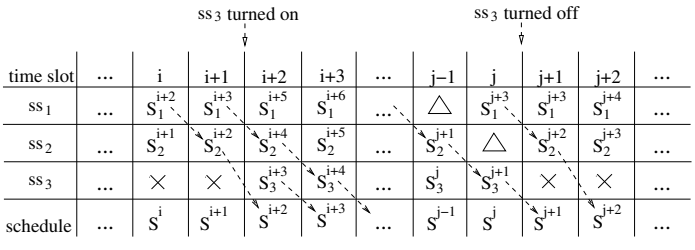


Fig. 7. An example of sub-schedulers being turned on and off.

ion: 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 threshold and does not bounce back in a 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 that the sub-scheduler is off, and a "Δ" means that 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. In the on state, a packet arrives at the input port every time slot. In the off state, no packet arrives. The packets arrived during an on state is called a burst and have the same destination. The average burst length is 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 [19]. The evaluated OpCut switch is of size 64×64. We have done simulations on 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 also implemented the basic non-pipelined scheduler, as well as the pipelined iSLIP that pipelines i sub-schedulers, each of which executes one iteration of iSLIP in a time slot. It is the straightforward way to pipeline iterative-matching-based algorithms, but 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 small 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. However, For p-iSLIP schedulers, the ratio drops sharply with the increment in traffic intensity. For all 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.

B. Average Packet Delay

Fig. 9 shows the average packet delay of the OpCut switch under different schedulers and traffic models. The ideal output-queued (OQ) switch is implemented to provide the lower bound on average packet delay. It can be seen that p-2SLIP and p-8SLIP perform very poorly due to underutilization of bandwidth caused by the duplicate 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 0.94 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, the system throughput can be improved through pipelining. In fact, except for the case under light uniform Bernoulli traffic where the extra delay introduced by pipelining is comparatively significant, the performance of p-k4-2SLIP is very close to that of np-8SLIP, which is in turn very close to that of the OQ switch in terms of average packet delay. That is, the 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.

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, as Fig. 10(b) shows. All schedulers, pipelined or not, are assumed to run 1SLIP. The average packet delay is sampled every 100 time slots. It can be seen that while non-pipelining and 2-subscheduler pipelining fail to sustain heavy traffic, and 3-subscheduler pipelining suffers from the 3 time-slot pipelining delay , adaptive pipelining 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. 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 provided a mechanism to pipeline the 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 an 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 illustrated that the pipelining mechanism effectively reduces scheduler complexity while maintaining good system performance.

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.

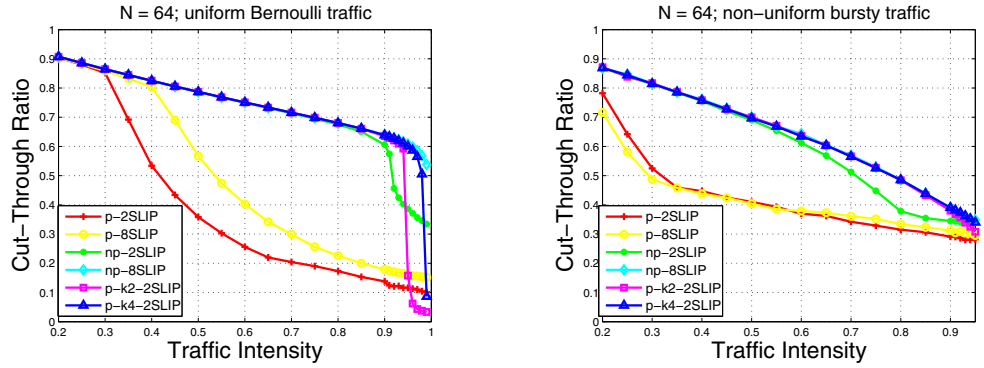
[6] F. Xia, L. Sekaric and Y. Vlasov, "Ultracompact optical buffers on a silicon chip,", *Nature Photonics* **1**, 2007.

[7] 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.

[8] 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.

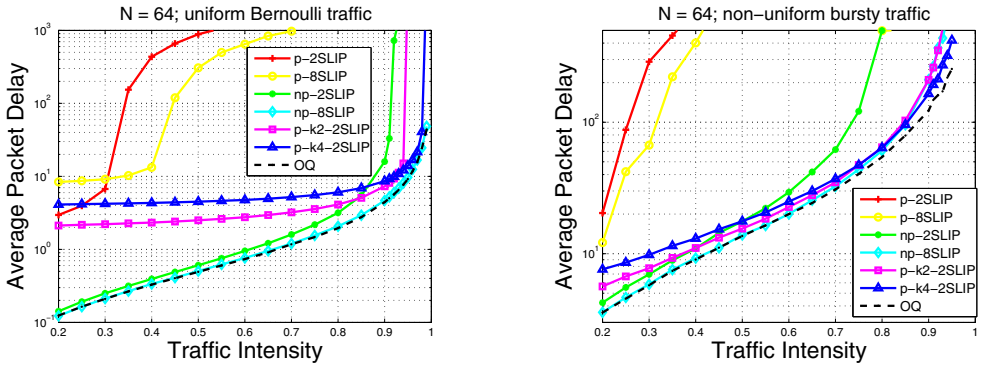
[9] C.-S. Chang, Y.-T. Chen and D.-S. Lee, "Constructions of optical FIFO queues," *IEEE/ACM Trans. Networking*, vol. 14, pp. 2838-2843, 2006.

[10] 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.



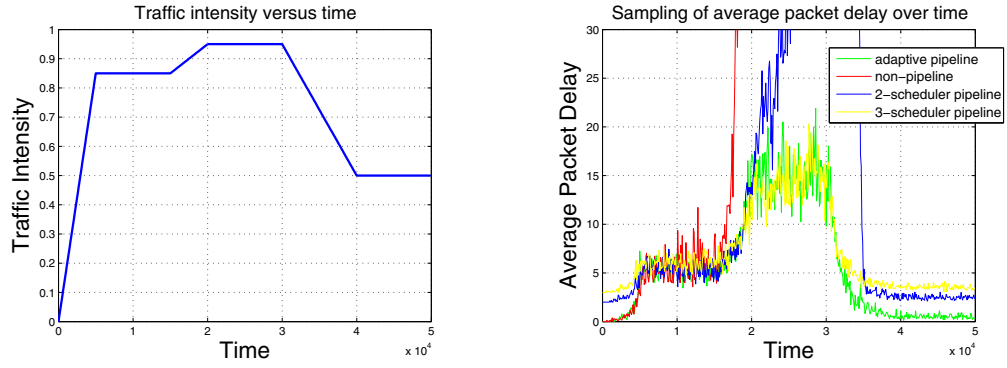
(a) (b)

Fig. 8. Packet cut-through ratio with non-pipelined and pipelined schedulers under different traffic models. 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.



(a) (b)

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



(a) (b)

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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.