PBC: A Partially Buffered Crossbar Packet Switch

Lotfi Mhamdi, *Member*, *IEEE*

Abstract—The crossbar fabric is widely used as the interconnect of high-performance packet switches due to its low cost and scalability. There are two main variants of the crossbar fabric: unbuffered and internally buffered. On one hand, unbuffered crossbar fabric switches exhibit the advantage of using no internal buffers. However, they require a complex scheduler to solve input and output ports contention. Internally, buffered crossbar fabric switches, on the other hand, overcome the scheduling complexity by means of distributed schedulers. However, they require expensive internal buffers—one per crosspoint. In this paper, we propose a novel architecture, namely, the Partially Buffered Crossbar (PBC) switching architecture, where a small number of *separate* internal buffers are maintained per output. Our goal is to design a PBC switch having the performance of buffered crossbars and a cost comparable to that of unbuffered crossbars. We propose a class of round robin scheduling algorithms for the PBC architecture. Simulations results show that using as few as eight buffers per fabric column and irrespective of the number N of input ports of the switch, we can achieve similar performance to buffered crossbars that use N buffers per fabric output.

Index Terms—Crossbar fabrics, partially buffered crossbars, scheduling.

1 Introduction

Interconnection networks constitute the basic infrastructure of a wide spectrum of computer and communication systems. Advances in computing systems (multicore processors, chip multiprocessor, clustering, etc.) as well as communication systems (the Internet) have created major engineering challenges of scale, speed, throughput, and efficiency of the underlying interconnection network. As a result, the growing performance requirements are increasingly pushing these systems to abandon traditional bus-based architectures and move toward packet-switch-based interconnects.

Numerous proposals for identifying suitable architecture for high-performance packet switches have been investigated and implemented both in academia and industry [1], [2], [3], [4]. These architectures can be classified based on various attributes such as queuing schemes, scheduling algorithms, and/or switch fabric topology. The crossbar-based architecture is the dominant architecture for today's high-performance packet switches because of its low cost and scalability. As a result, the vast majority of the commercially available core switches/routers are based on crossbar fabric with virtual output queuing (VOQ) [5]. The crossbar fabric architecture can mainly be classified into two categories: unbuffered or internally buffered crossbar fabric switch.

Extensive research work has been dedicated to unbuffered crossbar switches for more than two decades. Fig. 1a depicts an Input-Queued (IQ) crossbar fabric switch with VOQs at the inputs. The crossbar of an IQ switch runs at the same speed as external input/output ports (although, in practice, a speedup of 2 is generally used). In order to maintain this low-bandwidth requirement, an unbuffered IQ switch requires a centralized scheduler to resolve two main blocking problems, namely, input and output contention. Input contention results from the constraint that an input can send at most one packet every time slot. Similarly, output contention arises from the constraint that an output can receive at most one packet every time slot. These blockings make the task of the scheduler complex and the packets delay unpredictable. As a result, the switch performance essentially depends on its scheduling algorithm. Different classes of scheduling algorithms have been proposed [6], [7], [8], [9], [10]. Unfortunately, for high-bandwidth IQ switches, almost all scheduling algorithms are either too complex to run at high speed or fail to exhibit satisfactory performance. This is mainly attributed to the centralized design of these schedulers and the nature of the unbuffered crossbar switching architecture.

In order to overcome the scheduling complexity faced by IQ unbuffered crossbar switches, buffered crossbar switches have been proposed [11], [12]. Fig. 1b depicts a buffered crossbar switch, a crossbar where a limited amount of memory is added per crosspoint. The existence of internal buffers relaxes the output contention constraint, making the scheduling task much simpler. Buffered crossbars use distributed and independent schedulers (one per input/output port) to switch packets from the input to the output ports of the switch. A scheduling cycle consists of input scheduling, output scheduling, and flow control to prevent internal buffer overflow. Efficient scheduling algorithms have been proposed for this architecture [13], [14], [15]. The scheduling simplification comes at the expense of a costly crossbar. The crossbar has to contain N^2 internal buffers, where N is the number of input/output ports of the switch. The number of internal buffers grows quadratically with the

• *The author is with the Computer Engineering Laboratory, EWI Department, Mekelweg 4, 2628 CD Delft, The Netherlands. E-mail: lotfi@ce.et.tudelft.nl.*

Manuscript received 10 Apr. 2008; revised 2 Mar. 2009; accepted 11 Mar. 2009; published online 30 Mar. 2009.
Recommended for acceptance by G. Constantinides.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-04-0157.
Digital Object Identifier no. 10.1109/TC.2009.65.

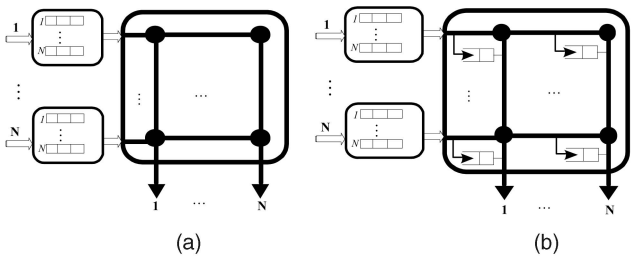

Fig. 1. Crossbar fabric variants. (a) Unbuffered crossbar fabric. (b) Buffered crossbar fabric, with N^2 internal buffers.

switch size and linearly with round-trip delays [4]. This makes buffered crossbar switches highly expensive, and hence, less appealing.

In this paper, we propose a novel architecture named the Partially Buffered Crossbar switching, where a small number of internal buffers ($B \ll N$) is maintained per fabric column instead of N. The PBC is designed to be a switching architecture having the performance of buffered crossbars but with a cost comparable to unbuffered crossbars. The PBC switch, depicted in Fig. 2, contains a small number of separate internal buffers, $B \ll N$, per fabric column. We propose a class of pipelined scheduling algorithms for the PBC switch and study their performance under various traffic patterns. The input and output schedulers are all embedded within the buffered crossbar fabric, reducing the required flow control signals from N^2 to $O(N\log N)$ [16]. The experimental study shows that setting the number of internal buffers per output to $B = 8$ is sufficient for the PBC to achieve optimal performance irrespective of the switch size N. Previous work proposed similar switching architecture to the PBC switch [17], [18], [19], [20]. Our work differs from previous art both at the architectural and the scheduling level. While the proposed architectures in [17], [18], [19], [20] rely on internal shared memory per output port, our PBC architecture uses separate internal buffers per output, and hence, avoids the requirement of expensive shared buffers. Second, the architecture proposed by [18], [20] was targeting multistage switches, whereas our proposed architecture targets single-stage switches. On the scheduling level, our proposed algorithms outperform those proposed by [19], [20] as will be shown in Section 4 of this paper.

The remainder of this paper is organized as follows: Section 2 presents the PBC architecture and its scheduling. In Section 3, we introduce our set of scheduling algorithms. The first algorithm is called Distributed Round Robin (DRR). It is based on round robin grant and credit schedulers per input and output port, respectively. Because of the credit release delay experienced by DRR, we propose an alternative algorithm named DROP that drops the unaccepted grants every time slot, and consequently, minimizes the credit release delay. We also propose an enhanced version of the DROP algorithm, named DROP-PR, that selects grants based on output priority. We propose a hardware implementation of the DROP algorithm. Section 4 presents the performance study of our algorithms under various settings. Finally, Section 5 concludes the paper.

2 The Partially Buffered Crossbar Switching Architecture (PBC)

This section introduces the Partially Buffered Crossbar switching architectural organization along with its scheduling.

2.1 Switch Model

We consider the Partially Buffered Crossbar switching architecture (PBC) depicted in Fig. 2. The switch operates on fixed sized cells. Variable size packets are segmented into fixed sized cells while inside the switch and reassembled back to packets upon their exit. The PBC has N input and N output ports. Each input port contains N VOQs, one per output port. When a cell, destined to output j, arrives at input i, it gets queued in $VOQ_{i,j}$. For every arrived cell, the input card sends a $\log N$ bit signal to the buffered crossbar chip to notify the input scheduler IS_i (residing in the

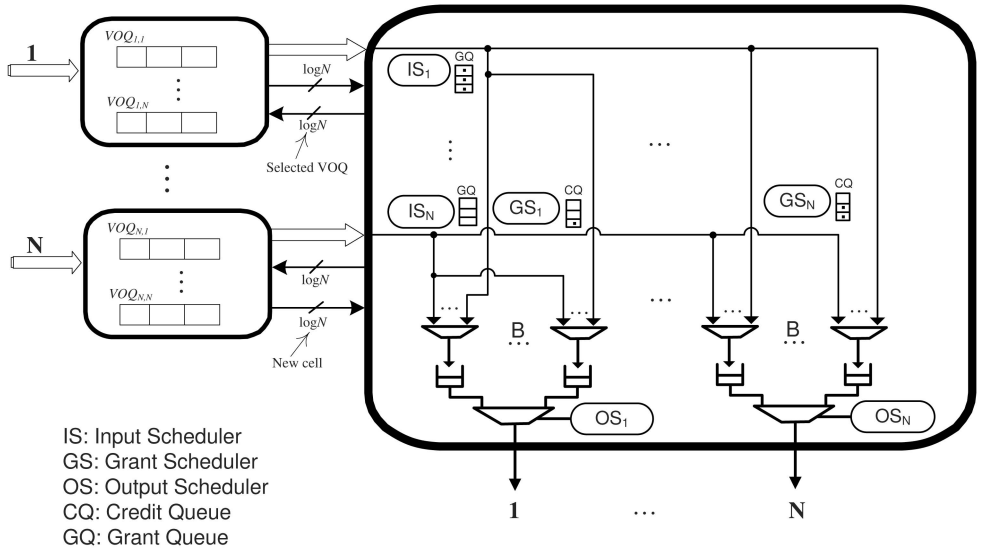

Fig. 2. The PBC switching architecture.

buffered crossbar) about the arrival of the new cell to $VOQ_{i,j}$. Once the input scheduler selects a cell for transfer from the input port to an internal crossbar buffer, it sends a $\log N$ bit signal to the input card indicating the selected VOQ.

The crossbar fabric contains a small number of internal buffers. These internal buffers are maintained per fabric output port and there are $B \ll N$ separate internal buffers per fabric output. All the schedulers are embedded within the buffered crossbar chip. There are N input schedulers (ISs), one per crossbar input. Each IS, i, controls the transfer of cells from the input line card, i, to the internal fabric buffers. When a new cell arrives to $VOQ_{i,j}$, the index of $VOQ_{i,j}$ is forwarded to IS_i inside the fabric chip. Input scheduler IS_i maintains a record[1] of the arrivals and departures of cells to and from the VOQs in input i and updates its record accordingly. The input scheduler decision is communicated back to its corresponding input by a $\log N$ bit signal indicating the index of the selected VOQ. The input scheduler decision is coordinated with a grant scheduler that manages the internal buffers availability for each output. The input and grant schedulers communicate via a grant queue (GQ) maintained per input scheduler. There are N GQs, one per input scheduler, and each contains N entries, one per output. When a grant scheduler (GS) j sends a grant g to input scheduler i, $GQ_{i,j}$ is set to one. Once input scheduler i accepts g, $GQ_{i,j}$ is reset to zero. Each GS contains a credit queue CQ that records the number of available internal buffers belonging to that GS. Each fabric output contains an output scheduler (OS) that arbitrates cell departure from the internal buffers to the output queue. There are N credit queues (CQs), one per output. Each CQ contains B entries and CQ_j records the availability of the internal buffers belonging to output j. A CQ is decremented whenever a grant is sent to the input and incremented during output scheduling. The scheduling process is described in details in Section 2.2.

Unlike a traditional CICQ switch that requires N^2 internal buffers, the PBC maintains only few internal buffers B per crossbar output, where $B \ll N$. These internal buffers are separate and run at the same bandwidth as the external line rate and they are used in order to maintain low bandwidth and avoid the need for high-throughput "shared" internal buffers. However, maintaining B separate buffers per output mandates the use of up to B.(N-to-1) parallel multiplexers per output. We conjecture that the cost and feasibility of the PBC (with B.N separate internal buffers and N.B.(N-to-1) parallel multiplexers) is lower than that of traditional CICQ design (with N^2 internal buffers and N.(N-to-1) parallel multiplexers), for the following reasons: 1) The crossbar chip is bound by I/O count and power consumption and not by the crosspoints logic, implying the underutilization of the crossbar chip die [16], [21]; 2) using the crossbar chip extra (unused) logic for either N^2 distributed or B.N shared internal memories running B times the external line speed results is excessive power consumption and can be costly; and 3) on-chip wires are inexpensive [22], and therefore, meeting the bandwidth goal can be achieved by adding more wires and

1. Depending on the implementation, it could be a table with N entries, one per VOQ.

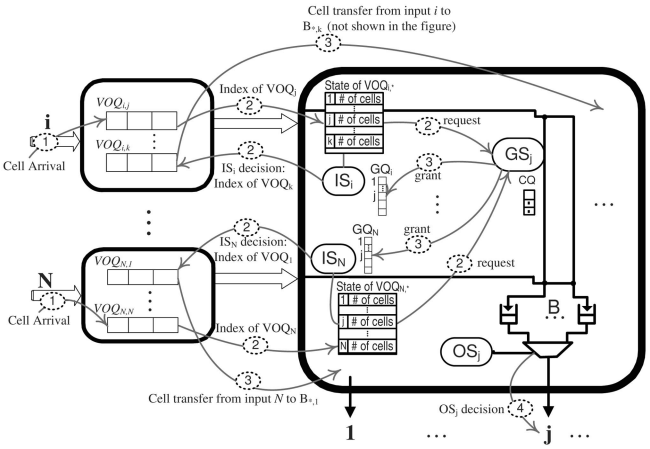

Fig. 3. The dynamic of the PBC switch. The succession of events, during one time slot, is illustrated by arrows and circulated numbers indicating the chronological order of each event. The input scheduler decision (event 2 from crossbar to input card) and the cell transfer (event 3 from input card to crossbar) are happening simultaneously with the request and grant events (events 2 and 3 inside the crossbar). Note that the input schedulers (IS_i and IS_N) decisions ($VOQ_{i,k}$ and $VOQ_{N,1}$) are based on past grant outcomes, excluding the current shown grant decisions ($VOQ_{i,j}$ and $VOQ_{N,j}$).

multiplexers in parallel. Consequently, embedding the N-input-, N-grant-, and N-output-distributed schedulers within the PBC chip is well within budget given the sufficient area on chip. Additionally, the PBC requires less flow control pins than traditional CICQs. The flow control signals between the input ports and the PBC core are shown in Fig. 2. In total $2N\log N$, flow control signals are used, in contrast to a traditional CICQ that require N^2 bit signals [12], [23], [24].

2.2 Scheduling Process

The scheduling process in the PBC switch is a combination of unbuffered as well as buffered crossbar scheduling. A scheduling cycle consists of input scheduling and output scheduling phases as in buffered crossbars. The input scheduling phase resembles a scheduling cycle in unbuffered crossbars, as it is based on request-grant-accept handshaking protocol. The input scheduling phase works as follows: During time slot t, each input scheduler IS_i checks its record (table of N entries representing the state of $VOQ_{i,*}$ and used for request phase, see Fig. 3) and sends a request (for every nonempty $VOQ_{i,j}$) to the GS at crossbar output j. Subject to internal buffers availability (CQ_j) and the grant scheduler policy, a grant may be sent back to the input scheduler i and stored in its GQ ($GQ_{i,j}$ set to 1). At the same time, input scheduler (IS_i) selects (by sending a $\log N$ bit signal to input i) a VOQ Head of Line (HoL) cell to be transferred to the internal buffers based on its GQ, excluding the current grants of time slot t. Meaning that the outcome of GS_j at time t is only valid during time slot $t+1$ or later. Fig. 3 depicts the succession of events during a time slot. We can see that the input scheduler IS_i decision (event 2 from the crossbar to input i) as well as cell transfer from the input to the internal buffers (event 3 on the arrow from input i to the internal buffers of output k) is performed at the same time as the request-grant phases (events 2 and 3 inside the

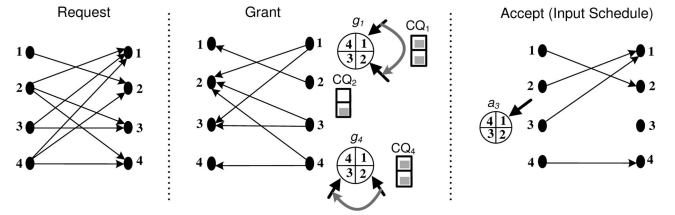

Fig. 4. A PBC input scheduling cycle, 4×4 PBC switch with $B = 2$. The grant pointers g_1 and g_4 are shown with their credit queues. Since each CQ contains two credits, each of g_1 and g_4 sends two grants and updates its pointer accordingly. Internal buffers B of output 1 receive two cells simultaneously because $CQ_1 = 2$.

crossbar). This means that the outcome of the grant decision (grant event 3 to input i) is excluded from the current input scheduling decision. Decoupling the input scheduling from the grant scheduling allows a two-stage pipelined arbitration, avoiding the need for synchronized coordination between the grant and the input (accept) schedulers on a time slot basis as does iSLIP [6] and PIM [25].

In traditional CICQ design, each of the N^2 internal buffers is dimensioned to cater for up to one round-trip time (RTT) worth of cells. The RTT is defined as the time from when a cell leaves the input port until a flow control signal is received back by the input. This implies that the amount of buffering per output is required to be $\geq N.RTT$. With the PBC, the situation is different. First the amount of buffering per output is required to be just $\geq B.RTT$, where $B \ll N$. Second, the RTT is defined differently in the PBC. Since the PBC employs a request-grant protocol, an input scheduler request must first secure a grant from the grant scheduler before it can proceed to transfer a cell. The grant scheduler, in turn, must first ensure the availability of credits before issuing grants. The RTT is therefore defined as the time from when a credit is generated until it is consumed by an input port and its associated cell is transferred to the internal buffer. Note that with our PBC request-grant protocol, only one RTT window is enough to achieve full-line rate to any requesting input. This is because of the shared CQ maintained by each grant scheduler and controlling the access from all inputs. However, a traditional CICQ switch would require N RTT windows to sustain full-line rate. This is attributed to the (per input) distributed output crosspoints, where an input is oblivious to the rate of the other inputs.

Because the number of internal buffers B maintained per output is much smaller than the number of competing inputs, N, crucial care to consider how to service cells during output scheduling is important. The internal buffers are separate and cells from the same VOQ may arrive to different internal buffers during consecutive time slots. In this case, we have to maintain in-sequence cell delivery. To this end, we employed a First-Come-First-Serve (FCFS) output scheduling[2] to ensure in-order cell delivery [26]. A cell departure from the internal buffers at output j causes CQ_j to increment by one. A cell arrival to an internal buffer at output j causes CQ_j to decrement by one. Likewise, the grant queue, at an input i, is incremented whenever a grant scheduler sends a grant to input i, and decremented whenever a cell departs the input port i.

2. It is possible to replace the output scheduling by imposing order on the writes to and reads from the internal buffers such as writes are from the right and reads are from the left.

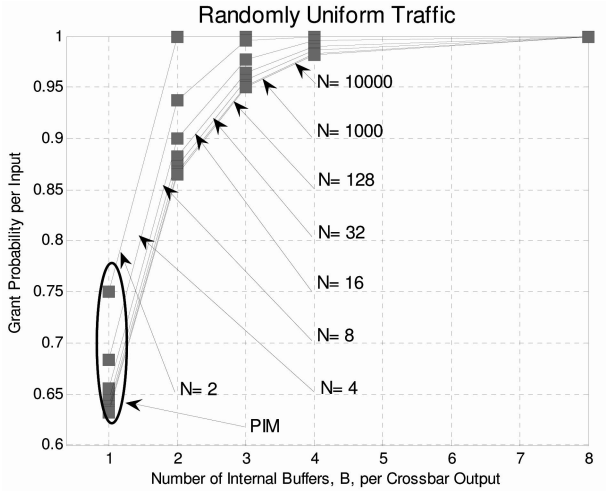

Fig. 5. Grant probability as function of switch size N and different internal buffer settings.

The input scheduling in PBC is similar to the matching performed by unbuffered crossbar scheduling. However, maintaining a small number of internal buffers makes it significantly different. The absence of internal buffers in an unbuffered crossbar switch meant that a grant arbiter can grant at most one input, to avoid output contention. Similarly, an input accept arbiter has to accept at most one grant, to avoid input contention. The PBC scheduling, while keeping the input contention constraint enforced, relaxes the output contention constraint by allowing conflicting cells (up to B) to be admitted to the internal buffers for the same output. This is equivalent to unbuffered crossbars schedulers accepting one grant and storing other $B - 1$ grants instead of discarding all the rest. Unbuffered crossbar schedulers resort to multiple iterations to improve the match size.

Fig. 4 describes a PBC input scheduling cycle. As we can see, the grant scheduler at output 1, g_1, sends two grants (to inputs 1 and 3) because its credit queue CQ_1 has two available credits (represented by two gray squares in CQ_1). The same process happens with g_3 and g_4. However, g_2 sends only one grant because its output buffers have only one location free ($CQ_2 = 1$). From the example, we can see the benefit in using internal buffers that allow conflicting cells to enter the fabric, thereby improving the grant opportunities per output. Consequently, the accept phase produces a bigger match size. Recall that for unbuffered crossbars employing a random scheduling policy with one iteration such as PIM [25], the probability that an input will remain ungranted is $(\frac{N-1}{N})^N$, where N is the port count of the switch [6]. As N increases, this probability tends to $\frac{1}{e}$. If we use the same random scheduling policy in the PBC with B internal buffers per output and assuming that packets are flushed in every time slot (memoryless Markov process), the probability that an input remains ungranted is $(\frac{N-B}{N})^N$. With increasing N, this probability tends to $\frac{1}{e^B}$ (almost 0 for $B \geq 4$). Fig. 5 illustrates this behavior. When $B = 1$, the PBC behaves identically to the bufferless PIM algorithm and the grant probability approaches 63 percent with increasing

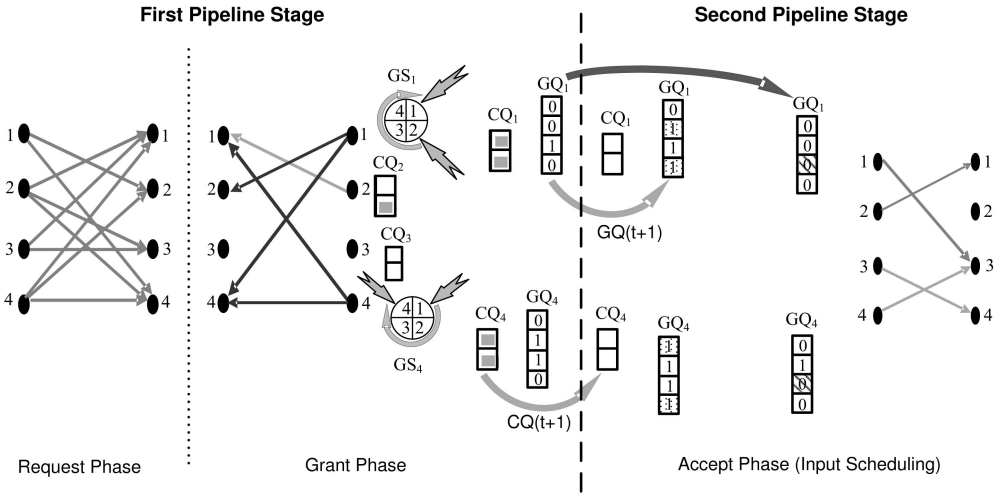

Fig. 6. A DRR input scheduling cycle for a 4×4 PBC switch with $B = 2$. Note that the input scheduling decision (hashed entries in GQ_i) is based on the GQ(t), while the outcome of the grant decision (dotted entries in GQ_i) will become valid from the next time slot $(t + 1)$ onward.

switch size. A small increase in B, just 2, scales up the grant probability to more than 86 percent for all switch sizes. When B is set to 4 per fabric output, the grant probability is 100 percent.

3 Scheduling in PBC Switches

This section introduces our set of scheduling algorithms for the PBC switching architecture. We propose a class of round-robin-based input scheduling algorithms. The output scheduling we use here and throughout this paper is based on FCFS policy, as discussed in the previous section. Each time, the output scheduler, at output j, performs its FCFS selection and sends a cell to the output queue, it increments CQ_j by one. As for the input scheduling, we propose a set of round-robin-based schedulers. The first algorithm we propose is named DRR and is described in the following section.

3.1 The DRR Algorithm

The DRR algorithm is similar to the scheme proposed by [19] and its grant scheduler's pointer update is the same as iSLIP [6]. This is because a grant sent by a GS to an input scheduler, if not immediately accepted, is stored and will eventually get accepted in the short term (less than N time slots later). The DRR differs in the way it assigns cells to internal buffers when they leave the input VOQs. However, this is specific to the PBC architecture. Cell assignment to internal buffers can be realized by maintaining a separate field in each entry of the GQ. Whenever GS_j grants to input i, it sets the entry $GQ_{i,j} = 1$ and the separate field (for internal buffer) to the index of the next free internal buffer $B_{k,j}$.

Algorithm 1. DRR
Grant Phase:
For each output, j, do

- While there are credits in CQ_j do
  - Starting from the grant pointer g_j index, send a grant to the first input, i, that requested this output (set $GQ_{i,j} = 1$).
  - Decrement CQ_j by one. /** CQ_j is incremented by one during output scheduling phase. **/
  - Move the pointer g_j to location $(i + 1)(mod\ N)$.

Input Scheduling Phase:
For each input, i, do

- Starting from the input pointer a_i index, select the first nonempty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
- Set $GQ_{i,j} = 0$.
- Move the pointer a_i to location $(a_i + 1)(mod\ N)$.

The DRR algorithm performs its arbitration, as described in Section 2.2. All the processing is done inside the buffered crossbar chip, by having the requests stored in per input tables with N entries, one for each output (refer to Fig. 3 for more details). Fig. 6 illustrates a DRR input scheduling phase. The first pipeline stage of the algorithm starts as follows: Based on the VOQs requests (request phase) and the CQ, each grant scheduler performs its arbitration. As shown in Fig. 6, GS_1 and GS_4 can each issue two grants because their credit queues have available credits ($CQ_1 = 2$ and $CQ_4 = 2$). GS_1 receives requests from inputs: 2, 3, and 4. It grants to inputs 2 and 4 because its pointer is at position 1 and there is already a pending grant for input 3 (third entry of $GQ_1 = 1$). After granting inputs 2 and 4, GS_1 points to input 1, as in iSLIP. GS_4 does the same, by granting to inputs 1 and 4, respectively (no grants for inputs 2 and 3 because their entries in $GQ_4 = 1$). However, GS_2 and GS_3 perform differently in this example because of their credit queues. GS_2 grants only to input 1 because its credit queue contains only one credit $(CQ_2 = 1)$. The other credit of output 2 is held by input 4 (second entry of $GQ_4 = 1$). GS_3 cannot issue any grants because it has no credits $(CQ_3 = 0)$ and its credits are held by inputs 1 and 4 (third entry of $GQ_1 = 1$ and third entry of $GQ_4 = 1$). The outcome of the grant scheduler will not be taken into account by the input scheduler until the next time slot. This is shown in Fig. 6 by dashed entries in GQ_1 and GQ_4 (second entry of GQ_1 and last entry of GQ_4). Simultaneously, with the first pipeline phase, the second pipeline phase is executed as follows: Based on the grant queues so far, each input scheduler i selects the next nonzero entry j of its GQ_i and sends the corresponding HoL cell of $VOQ_{i,j}$

to the internal buffer. It also updates its GQ, by resetting the entry of the sent cell (both third entries of GQ_1 and GQ_4 in accept phase are reset). Then, it increments its pointer by one Mod (N). We can see in this example that $B_{3,*}$ (corresponding to output 3) receives two cells simultaneously, this is the advantage of the PBC architecture—allowing conflicting cells to enter the fabric. Note that the accept phase in Fig. 6 is only for input scheduling whereby the speedup of one constraint is always ensured. For example, the cells sent by $VOQ_{1,3}$ and $VOQ_{4,3}$ will be queued in $B_{3,i}$ and $B_{3,j}$ such that $i \neq j$.

3.2 The Credit Release Delay

The DRR scheme experiences the same credit release delay as with the scheme in [19]. Credit release delay arises when multiple grant schedulers grant to the same input concurrently. Because DRR returns credits one at a time (input contention), credits may not return fast enough. This, consequently affects the rate at which grants are sent back to other inputs, hence delaying the transfer of cells from the input line cards. In order to reduce this delay, a grant throttling mechanism was proposed in [19]. It consists of setting a threshold (TH) for the grant queue and requests from an input are eligible so long as the grant queue relative to their input is less than TH. While this mechanism speeds up the credits release, it does not completely eliminate it or minimize it. Additionally, it requires some extra signaling to control the grant queues thresholds.

Our solution to credits release delay is different. We do not want to just lower the credit release delay. Instead, our goal is to completely eliminate it or set it to its absolute minimum. First, we need to quantify this delay. A grant has to wait up to N time slots in each grant queue before its associated credit is released back. Since a grant arbiter can issue up to B grants simultaneously, an input request waits, therefore, at most $\frac{N^2}{B}$ time slots before it gets granted. To better explain this, consider the scheduling example depicted in Fig. 6. Input 3 has a request for output 3. However, output 3 has no credits because they are already held by inputs 1 and 4, respectively. So, in the worst case, output 3 will grant to input 3 after each credit is released by all other inputs (grant queue updated). This can take up to four time slots in each of the four grant queues. However, since we have two credits per credit queue, they are always divided among requesting inputs. Therefore, output 3 will grant to input 3 no later than $\frac{4^2}{2} = 8$ time slots since input 3 first issues its request.

When $B = 1$, the performance of DRR is similar[3] to iSLIP (see Figs. 14 and 15). However, as B increases, the credit release delay decreases ($\frac{N^2}{B}$ decreases). Thus, the problem of credit release delay is now reduced to solving the grant queuing delay. Minimizing the credit release delay means altering the grant mechanism to reduce the grant queuing delay. We, therefore, modified the way DRR allocates grants per input; hence, a new grant scheduler is proposed. Instead of *storing* the grants, which are not accepted, in a grant queue while they wait their turn to be accepted, and therefore, causing credits waiting (delayed) to get released, we simply *drop* them. The new devised scheme never stores grants, instead they are just dropped and the scheme is named DROP.

3. The slight difference observed results from the DRR input scheduler pointer update mechanism. Unlike iSLIP, DRR input scheduler pointer is fully desynchronized, as in [8].

3.3 The DROP Algorithm

Dropping the not accepted grants implies that the pointer updating scheme of the grant scheduler has to change. This is because, otherwise, using the iSLIP pointer updating mechanism results in pointer synchronization similar to RRM [6]. This convergence of iSLIP to RRM, under our settings, comes from the two-stage pipeline scheduling relative to the PBC architecture. Recall that DRR *stores* the unaccepted grants, guaranteeing their immediate or soon acceptance (within at most N time slots), and therefore, the grant pointer can safely be updated. This is similar to iSLIP, where the grant pointer is updated only on accept. With the DROP scheme, now, *dropping* the unaccepted grants, the grant pointers move becomes uncoordinated. This is because the grant pointers are updated in the first pipeline stage (time t), while the outcome of their grant (the accept decision) takes place in the second pipeline stage (time $t + 1$) during the accept phase. Therefore, by the time $t + 1$, we know which grant is accepted and which grants are rejected (dropped), it is already late to correctly reupdate the pointers which were updated in time t. Therefore, using the iSLIP pointer update mechanism in DROP means "blindly" updating the grant pointers, which leads to synchronization and poor performance as in RRM [6]. To overcome this problem, we use fully desynchronized grant pointer settings, similar to [8]. The grant pointers are initially set to different positions and are always incremented by one irrespective of the accept/drop outcome. Proceeding this way, a request waits no more than N time slots before it gets granted, due to the synchronous advance of the pointer. This is to be compared to $\frac{N^2}{B}$. The specification of the DROP algorithm is as follows:

Algorithm 2. DROP

Grant Phase:

All output pointers, g_j, are initialized to different positions. For each output, j, do

- Set CQ_j equals the number of nonfull internal buffers of output j.
- While there are credits in CQ_j do
  - Starting from g_j index, send a grant to the first input, i, that requested this output (set $GQ_{i,j} = 1$).
  - Decrement CQ_j by one.
- Move the pointer g_j to location $(g_j + 1)(mod\ N)$.

Input Scheduling Phase:

All input pointers, a_i, are initialized to different positions. For each input, i, do

- Starting from a_i index, select the first eligible $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
- Drop the remaining grants (reset GQ: $GQ_{i,*} = 0$).
- Move the pointer a_i to location $(a_i + 1)(mod\ N)$.

In order to describe the difference between the two pointer updating mechanisms, we used a 4×4 PBC switch with $B = 2$ with the same initial pointers settings. Fig. 7

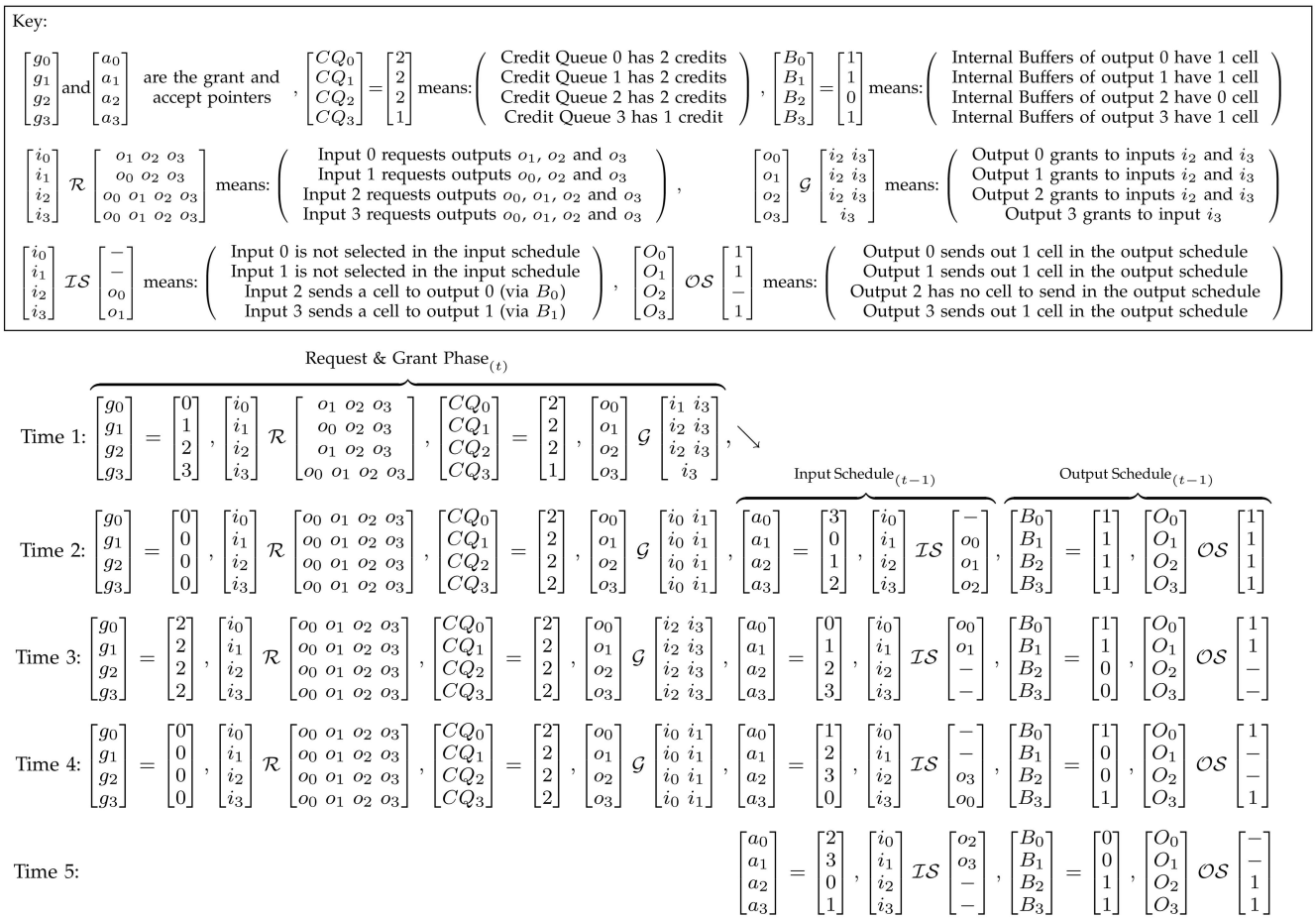

Fig. 7. DROP scheduling example using iSLIP-like pointers settings. At the end of Time 1, each grant pointer g_i moves to one position beyond its granted inputs. The DROP input scheduler processes the grants of Time 1 at Time 2 (second pipeline stage), accepts only one grant, and *drops* the rest. At the end of Time 1, pointers g_i become synchronized and alternate infinitely between granting inputs {0,1} and {2,3}, leading to a maximum throughput of only 50 percent.

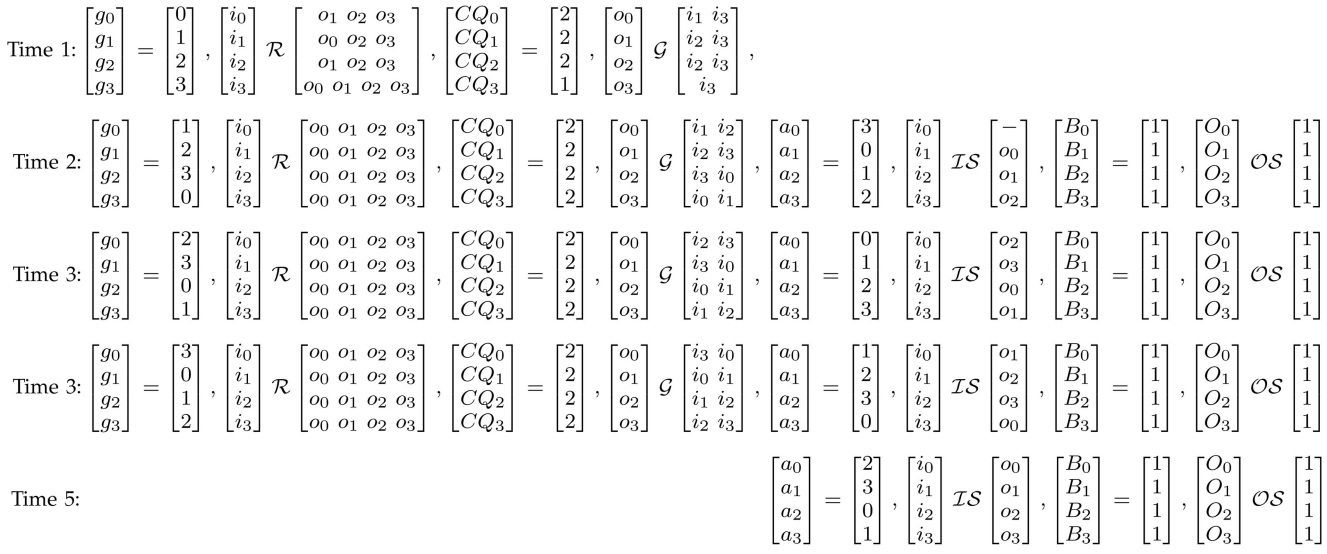

Fig. 8. DROP scheduling example using fully desynchronized pointer settings. Because the pointers are initially set to point to different positions and synchronously advance each time slot, they stay always desynchronized. These settings overcome the one time slot pipeline delay between the grant and accept decisions and avoid pointers synchronization. From Time 1 onward, the PBC delivers 100 percent throughput.

shows the iSLIP-like pointer updating mechanism, and Fig. 8 uses a fully desynchronized pointer updating scheme. Initially, the grant pointers g_i for both schemes are set at the same positions ($g_0 \rightarrow 0$, $g_1 \rightarrow 1$, $g_2 \rightarrow 2$, and $g_3 \rightarrow 3$). During Time 1, grant scheduler 0 (GS_0) in Fig. 7 performs as follows: g_0 starts from position 0 and searches for the first two inputs that request it (because it has two credits, $CQ_0 = 2$). The request phase shows that only inputs 1 and 3

are requesting output 0. So, GS_0 grants to inputs 1 and 3 and updates its pointer g_0 to position 0 (one position beyond 3). The other grant schedulers perform in the same way, based on their request vectors, and update their pointers. The accept decision (input scheduling) relative to the previous grant decision takes place at Time 2 (second pipeline stage), where input 1 accepts output 0, etc. Note that g_1 has granted to inputs 2 and 3 and despite *dropping* its grant to input 3, g_1 is now pointing to input 0. As a result, from Time 2 onward, the g_i pointers become synchronized and alternate indefinitely between pointing to inputs 0 and 2. This results in either granting only inputs 0 and 1 or only inputs 2 and 3. Consequently, no more than two cells per input scheduling cycle are transferred, leading to a maximum throughput of 50 percent. Fig. 8 uses the fully desynchronized pointers that always move by one position. These settings overcome the grant and accept pipeline delay and prevent pointers synchronization. We can see that four cells are always granted from four different inputs, resulting in the PBC delivering 100 percent throughput.

3.4 The Prioritized DROP Algorithm

We wanted to further improve the performance of the DROP scheme. The need to enhance DROP stems from two reasons. From one hand, DROP works in a two-stage pipeline meaning that it suffers some initial delay. On the other hand, DROP is designed for the PBC architecture that is assumed to have a limited small number of internal buffers per output, B. With these observations, we propose an enhanced version of DROP that services grants based on output priority. The idea is similar to the LOOFA algorithm previously proposed for input and output queued switches [27]. We call this version prioritized DROP and refer to it as DROP-PR. The specification of the DROP-PR scheme is as below.

When selecting a cell for input scheduling, DROP-PR takes into account the occupancy of the internal buffers belonging to an output. When a grant scheduler grants an input request, it sends back the grant with an additional priority bit P. The priority bit informs the granted input whether or not the grant comes from an output with empty internal buffers (prioritized output). During the input scheduling phase (second pipeline stage), the input scheduler first gives priority to grants where the priority bit is set to 1. This would give priority to idle outputs which, in turn, results in balanced internal buffers per column and higher throughput. The priority bit is obtained by logically OR-ing the entries of the CQ.

Algorithm 3. DROP-PR

Grant Phase:

All output pointers, g_j, are initialized to different positions. For each output, j, do

- Set CQ_j equals number of nonfull internal buffers of output j.
- Set the priority bit, P, to the logic OR of CQ_j entries.
- While there are credits in CQ_j do
  - Starting from g_j index, send a grant to the first input, i, that requested this output (set $GQ_{i,j} = 1$ and add bit P).
  - Decrement CQ_j by one.
- Move the pointer g_j to location $(g_j + 1)(mod\ N)$.

Input Scheduling Phase:

All input pointers, a_i, are initialized to different positions. For each input, i, do

- Starting from a_i index, select the first nonempty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and bit $P = 1$ and send its HoL cell to the internal buffer.
- If no HoL cell is selected, Then
  - Starting from a_i index, select the first nonempty $VOQ_{i,j}$ for which $GQ_{i,j} = 1$ and send its HoL cell to the internal buffer.
- Drop the remaining grants (reset GQ: $GQ_{i,*} = 0$).
- Move the pointer a_i to location $(a_i + 1)(mod\ N)$.

Since DROP_PR grants inputs based on their priority, starvation of input VOQs can occur. To see this, consider a 3×3 PBC with $B = 2$. Assume that at time t, all internal buffers of output 0 (B_0) are empty ($VOQ_{*,0}$ have high priority). Assume also that $VOQ_{i,0}$ is empty, while other VOQs are nonempty and arrivals to input 0 occur continuously to $VOQ_{0,0}$ from $t + 1$ onward. In this scenario, the two other VOQs of input 0 ($VOQ_{0,1}$ and $VOQ_{0,2}$) will starve indefinitely. This is because the priority bit of $VOQ_{0,0}$ is set to 1, while the priority bits of $VOQ_{0,1}$ and $VOQ_{0,2}$ are each set to 0. We can overcome this problem as follows. Whenever the accept pointer gets incremented from j to $j + 1$, the priority bit of $VOQ_{i,j}$ is also checked and if it is equal to zero, it gets set to "1." This will ensure that $VOQ_{i,j}$ will be serviced in the next round of the accept pointer (within at most N times slots). Additionally, $VOQ_{i,j}$ will have the lowest priority among the queues with priority value "1" because the pointer is at position $j + 1$. At the same time, $VOQ_{i,j}$ has the highest priority among the queues with a "0" priority bit. When this process is applied to all VOQs of an input, it avoids the starvation problem and, at the same time, ensures the fairness among all VOQs.

3.5 Hardware Implementation

In this section, we propose a possible hardware implementation of the DROP[4] grant scheduler. The design assumes a 32×32 PBC with four internal buffers per output ($B = 4$). This means that the grant scheduler can *concurrently* grant up to four requests in a single time slot. Fig. 9 depicts a block diagram of the proposed design. The circuit has three inputs and one output. The inputs are: 1) a 32-bit request vector containing the requests; 2) a 32-bit vector (1-hot) representing the current position of the highest priority pointer; and 3) a 3-bit vector, called the available internal buffers vector B and contains the number of the available internal buffers belonging to the grant scheduler. The output of the circuit is a 32-bit vector containing the grant decision. The scheduler works in two phases as follows. First, the request vector is processed (starting from the highest priority pointer until the last position in the vector). Second, we wrap around from position 0 until reaching the position where we started in the first phase. These two phases are shown in Fig. 9 by two dashed rectangles, where the upper rectangle corresponds to phase 1. The two phases can be performed in parallel, with the second phase starting two clock cycles after the first.

4. The same design can be used for the DROP_PR scheduler if we append a priority bit to the output. The priority bit is a 3-inputs OR gate of vector B.

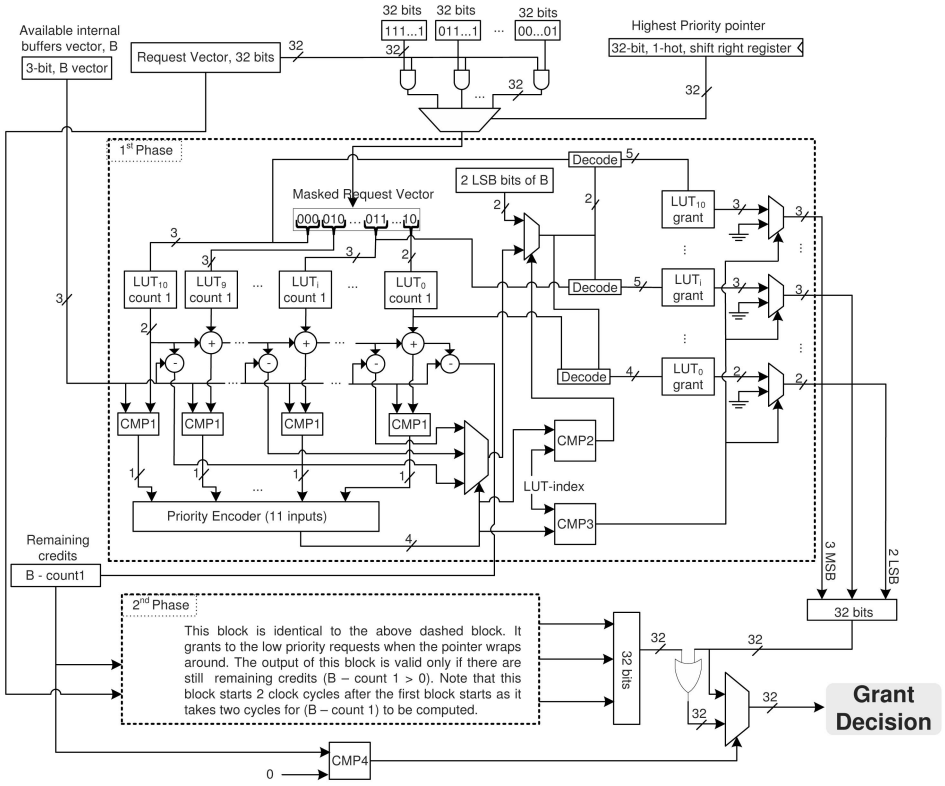

Fig. 9. Block diagram of the DROP algorithm.

The grant scheduler first determines the highest priority request. This is achieved by generating 32 request masks based on the 32-bit input request vector, and then, selecting the masked requests based on the position of the highest priority index. Note that since DROP uses a fully desynchronized pointer updating mechanism, the highest priority pointer can be designed as a 32-bit 1-hot shift right register. The masked request vector along with the B vector are used as inputs to the first phase of the scheduler. This phase works as follows: The 32-bit masked request is processed in parallel by splitting the 32-bit string into 10 entities (subrequests) of 3-bits each and one (last) entity with 2-bits. Each 3-bit string is fed to a Lookup Table (LUT) to count the number of 1s contained in it, as shown in Table 1. The counting of 1s is done in parallel by the 11 LUTs. The cumulative output of the LUTs (starting from the left, LUT_{10}) is compared to the value of B. The two inputs to each comparator $\mathrm{COMP1}_i$ are B and $\sum_i^{10}(\mathrm{count}'1_i)$. The output of $\mathrm{COMP1}_i$ is 0 if $\mathrm{B} > \sum_i^{10}(\mathrm{count}'1_i)$, 1 otherwise. This process results in an 11-bit string with the first "1" bit, from the left, indicating the LUT starting from which we have more 1s than B, and therefore, we can stop the grant process there. Simultaneously with this, the cumulative LUTs output is subtracted from the value of B. This will tell us later whether the LUT at which we satisfied the condition of (count 1) $\geq$ B is included in the grant and how many 1s should we take from it.

TABLE 1
Encoding of the Number of 1s in LUT-Count 1

Sub-Request	LUT Output
000	00
001	01
010	01
011	10
100	01
...	...
111	11

The LUT, i, at which we should stop is indexed by the output of a priority encoder (PE) that takes as input, the output of the 11 parallel COMP1 chain. Now, we need to pass all the values of the LUTs as they are starting from LUT_{10} and moving right until LUT_{i+1}. Depending on the subtraction, we may take only a subset of the 1s in LUT_i, and finally, we mask out (reset) the bits of the LUTs starting from LUT_i down to LUT_0. Because we do not know a priori where the index i will be nor how many bits should we take from it, we have to encode the LUTs depending on whether we should expect B or $0 < s < B$, where s is the opposite value of (B - count 1). We refer to these LUTs as LUT-grant with inputs being a 5-bit decoding (concatenation) of the 3-bit entities as above followed by either s or the 2-bit LSB of B.[5] The output of the LUT is a 3-bit value representing the grant bits based on the 3-bit entity input as well as the available credits (either s or B), as depicted in Table 2. We compare the index of every LUT to the index of the LUT equaling to the output of the PE (that is LUT_i). This is done using COMP2 that outputs 0 if the current LUT index $> i$ and 1 otherwise. This results in $\mathrm{LUT}_{10} \rightarrow \mathrm{LUT}_{i+1}$ encoded using B and $\mathrm{LUT}_i \rightarrow \mathrm{LUT}_0$ encoded using s. In order to ensure that LUT_i is encoded with s and the other LUTs to

5. The 2 LSB of the B vector are sufficient since we can encode 4 available Bs as "00." This is because the value "00" that would otherwise refer to 0 Bs are available can never happen due to output scheduling that ensures emptying at least 1 B every time slot.

TABLE 2
Grant Encoding in LUT-Grant

Input	LUT Output
000 **	000
001 00	001
001 01	001
001 10	001
...	...
011 01	010
...	...
111 10	110
111 11	111

the right are reset to 0, we use COMP3 that outputs 1 if current LUT index $< i$ and 0 otherwise. This will give the desired output.

The second phase of the scheduler, lower dashed rectangle in Fig. 9, grants to the rest of the requests by wrapping around from the last request index and back until the starting position of the highest priority pointer. This phase works exactly as the first phase above. The only two differences are: 1) phase 2 takes the (B - count 1) as input instead of B and 2) although the two phases are performed in parallel, phase 2 always has to wait two clock cycles in order to get the result of (B - count 1) from phase 1. If (count 1) $\geq$ B, then the grant decision is that of the first phase output, otherwise the outputs of both phases are logically OR-ed to get the output grant decision.

We implemented the design in reconfigurable logic, using the Xilinx Virtex II XC2V250 as the target device and the Xilinx ISE 9.1 design flow platform. Our design has been segmented into six clock cycles with a clock cycle time of 7.8 ns, equal to 46.8 ns per scheduling cycle. We compared our design to the Masked Priority Encoder (MPE) design proposed in [28] as well as the PROPOSED design described in [29]. Our design outperforms both the PME and the PROPOSED designs. The related work design assumes one iteration of the grant scheduler since it gives one grant at a time. However, in our case $\mathrm{B} = 4$, which is equivalent to the designs in [28], [29] using four iterations. Table 3 shows our results and those found in [28]. Note that the delay results in [28], [29] are, here, multiplied by 4 to account for multiple grants/iterations. We can see that we achieve the shortest delay among all designs. However, due to its parallel grant process, our design trades hardware resources for time and requires more area than the other designs. It is worth noting that our design can be easily reconfigured to account for $\mathrm{B} = 7$, since B is encoded using 3 bits. This would not cost extra delay, but might increase the area requirement since we have to use slightly bigger LUTs. If we compare our timing results with $\mathrm{B} = 7$ to related work designs with seven iterations, the difference gets much wider and our design would result in far shorter scheduling delay.

TABLE 3
Delay and Area Results, $\mathrm{N} = 32$, DROP with $\mathrm{B} = 4$, MPE and PROPOSED with Four Iterations Each

	DROP, B=4	MPE	PROPOSED
Delay (ns)	46.8	52	142
Area (slices)	431	355	150

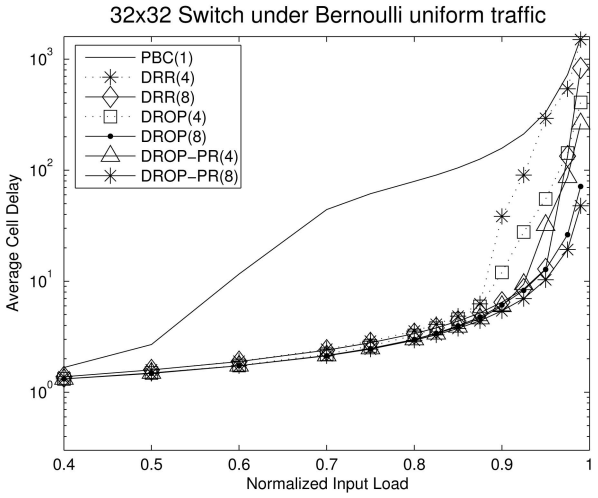

Fig. 10. Average cell delay of the PBC algorithms under Bernoulli uniform traffic.

4 Performance Results

This section presents the performance study of the PBC switching architecture. The study is aimed at comparing our proposed architecture to both the unbuffered and the buffered crossbar fabric architectures as well as an ideal Output-Queued (OQ) switch. The DRR algorithm is a slightly enhanced version of the algorithm proposed by [19], [20], where a fully desynchronized round robin pointer setting is used in the DRR accept phase, while a simple round robin was used in [19], [20]. Therefore, DRR's performance can be used to refer to that of the algorithm in [19], [20]. The experiments are carried out under four input traffic patterns: Bernoulli uniform, Bursty uniform, Diagonal traffic, and Unbalanced traffic. We tested different PBC switch sizes, each with different internal buffer settings. In this section, we mostly present the results of switch sizes of 16×16 and 32×32 only.

4.1 Uniform Traffic

Fig. 10 illustrates the average cell delay performance of each of our proposed algorithms under Bernoulli uniform traffic. We measured the delay of each of the algorithms with different internal buffer settings. When the number of internal buffers per output, $B = 1$, all the three algorithms have the same delay because there is no credit release delay. For this reason, we denote any of the algorithms by "PBC(1)" in the figure. Increasing B to as few as four internal buffers per output boosts the performance by an order of magnitude. The delay improvement is less sharp for B between 4 and 8. This behavior agrees with our model (refer to Fig. 5) discussed in Section 2.2. When $B = 4$ or more, the granting likelihood is almost 100 percent from each grant scheduler to each input scheduler. The same behavior is observed under Bursty uniform arrivals, as depicted in Fig. 11.

Assessing the performance of each of our three proposed algorithms (DRR, DROP, and DROP-PR, respectively) requires tuning different parameters such as switch size, internal buffers size, and input traffic loads, hence many plots. However, because we are most interested in switch cell delays under heavy input loads, in the following two figures (Figs. 12 and 13), we fixed the input load at

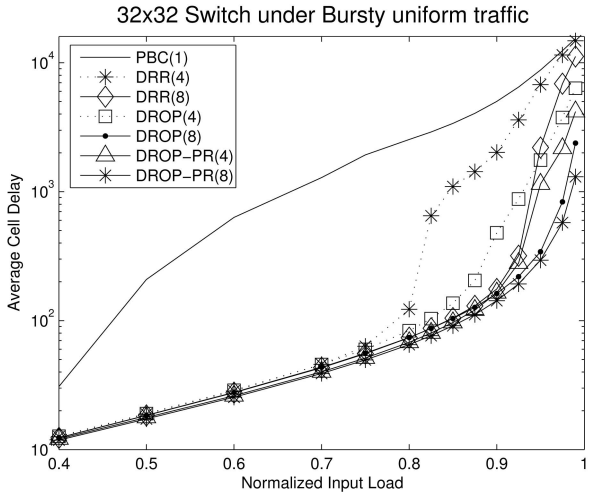

Fig. 11. Average cell delay of the PBC algorithms under bursty uniform traffic.

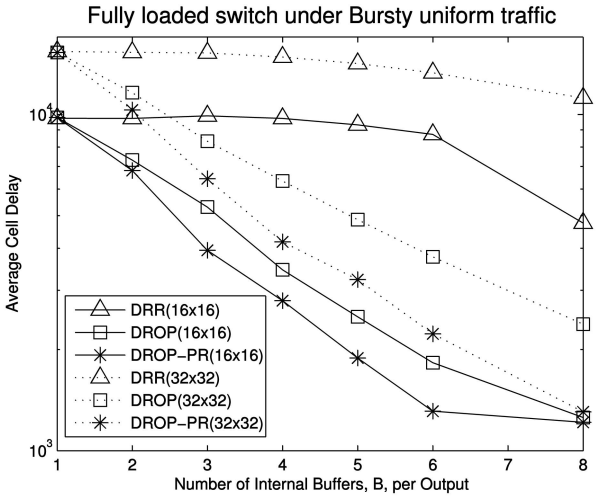

Fig. 13. PBC Performance under bursty uniform arrivals.

99 percent and varied the switch size as well as the number of internal buffers per output for each algorithm. Fig. 12 depicts the performance of each of our algorithms under Bernoulli uniform arrivals for a 16×16 and 32×32, respectively. We observed the average cell delay as a function of the number of internal buffers per output, B. We can see that when $B = 1$, unbuffered crossbar switch, the three algorithms have the same delay which is comparable to iSLIP with one iteration. This is because when $B = 1$, every request waits for the same time (N^2 times slots at most) before it gets served.

However, with increasing B, both DROP and DROP-PR have lower cell delays than DRR because of their fast credits release. Recall that the credit release delay (and consequently, grant and service delays) of DRR is $\frac{N^2}{B}$. However, both DROP and DROP-PR have a credit release delay of N. As B increases (especially as B approaches N, not shown in the Figures), all the algorithms have the same delay. However, we are only interested in PBC switches with $B \ll N$. The same trend is also observed under bursty arrivals (see Fig. 13). The delay of DRR decreases faster under Bernoulli uniform than under Bursty arrivals because of the burstiness effect. DROP-PR has the overall lowest delay because it prioritizes outputs with empty internal buffers, resulting in more balanced internal buffers occupancies, and hence, lower cell latencies. We also observed that under various switch sizes (8, 16, 32, and 64) running DRR, the delay when using $B = 3$ is greater than when using $B = 2$. We believe this to be related to the credit release delay experienced by DRR for these B values; however, further investigation is needed.

We compared the average cell latency of the DROP-PR algorithm for a 32×32 PBC switch to an unbuffered crossbar switch, a fully buffered crossbar switch, and an ideal OQ switch. The iSLIP algorithm is used for the unbuffered crossbar architecture. The fully buffered crossbar switch uses input round robin (RR) scheduling and the Oldest Cell First (OCF) output scheduling, for fair comparison with our proposed algorithms. The comparison is performed under uniform Bernoulli and Bursty arrivals. Fig. 14 depicts the performance of DROP with different internal buffers, iSLIP (with one and four iterations), RR_OCF, and OQ. Irrespective of whether the input traffic is Bernoulli or Bursty, DROP-PR(1) (1 refers to $B = 1$) has a similar behavior to 1SLIP, as described earlier (see Section 3). As B increases, the delay of DROP-PR significantly decreases. It approaches that of an ideal OQ with just eight internal buffers per output ($B = 8$). Similar performance is observed under bursty arrivals (Fig. 15). These results suggest that a PBC switch can replace a buffered crossbar, or even an ideal OQ switch with as few as eight internal buffers per output. These results can also afford a switch designer the choice depending on the constraints and core requirements. For example, if the delay-cost product is the main target, there is the option to replace an unbuffered crossbar switch employing 4SLIP with a PBC switch with only four internal buffers per output (see the delay performance of 4SLIP and DROP-PR(4) in Figs. 14 and 15, respectively). However, if performance is the main criteria with a little flexibility in cost, one can then employ a PBC switch with eight internal buffers per output since it exhibits ideal performance. This is better illustrated in Fig. 16, where we compared the DROP_PR(8) to OQ under different sizes of 8, 16, 32, and 64, respectively. We can see

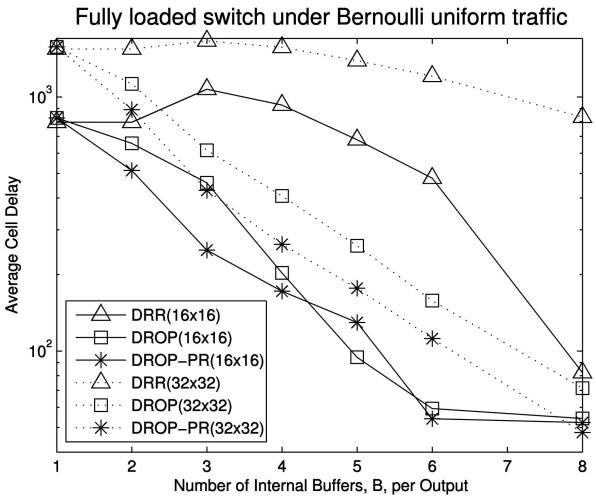

Fig. 12. PBC Performance under Bernoulli uniform arrivals.

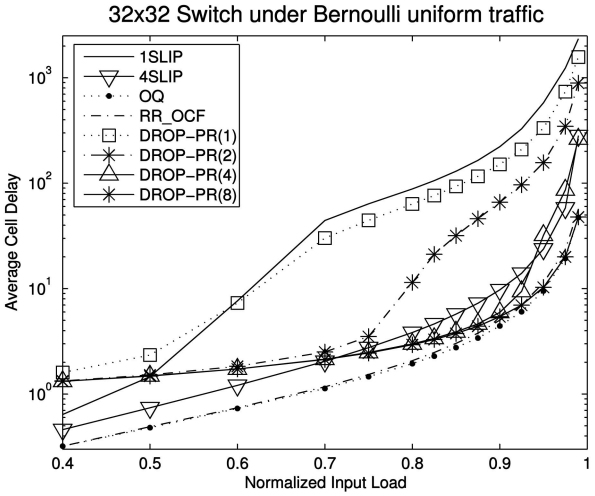

Fig. 14. Performance under Bernoulli uniform arrivals.

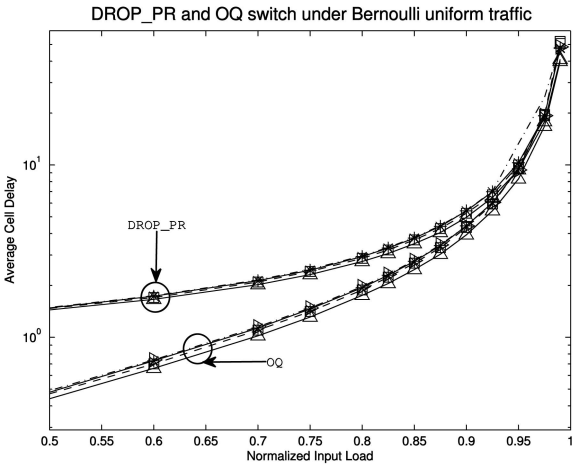

Fig. 16. Average delay comparison between DROP_PR and OQ under Bernoulli uniform traffic and different switch sizes ($\mathrm{N} = 8, 16, 32$, and 64).

that under heavy loads, both systems have the same delay irrespective of the switch size. Note that under light loads (≤ 80 percent), DROP_PR has relatively higher delays which is due to the pipeline delay.

4.2 Nonuniform Traffic

The performance analysis is carried out under two nonuniform traffic patterns: the diagonal traffic as defined in [30] and the unbalanced traffic as defined in [13]. These two traffic patterns are known to be hard to schedule and considered to be representative in assessing the performance of algorithms.

The unbalanced traffic is defined by using an unbalanced probability ω. For an $N \times N$ switch, the traffic load at each input port is defined by ρ. Then, for each input port s and output port d, the traffic load $\rho_{s,d}$ is given by

$$\rho_{s,d} = \begin{cases} \rho(\omega + \frac{1-\omega}{N}), & \text{if } s = d, \\ \rho\frac{1-\omega}{N}, & \text{otherwise.} \end{cases}$$

Note that when $\omega = 0$, the load is uniform over all outputs; and when $\omega = 1$, the traffic is totally unbalanced (only the diagonal). The Diagonal traffic is defined as in the following traffic matrix, for 4×4 switch:

$$\lambda(Diagonal) = \frac{1}{3}\begin{pmatrix} 2\rho & \rho & 0 & 0 \\ 0 & 2\rho & \rho & 0 \\ 0 & 0 & 2\rho & \rho \\ \rho & 0 & 0 & 2\rho \end{pmatrix}.$$

This is a very skewed and critical traffic, in the sense that input i has packets only for output i and output $|i+1|$. A diagonal load has $\lambda_{i,i} = \frac{2\rho}{3}$, $\lambda_{i,|i+1|} = \frac{\rho}{3}$ $\forall i$, and $\lambda_{i,j} = 0$ for all other i and j.

To further endorse our claims with respect to the PBC performance, we analyzed the stability of a 32×32 PBC switch under unbalanced traffic arrivals. We set the switch input load at 100 percent and varied the unbalanced coefficient ω and observed the switch throughput performance. Fig. 17 depicts the performance of the PBC switch with DROP-PR algorithm and different internal buffer settings and compares it to a buffered crossbar (using RR_OCF scheduling). We can see that with $B = 2$ internal buffers per output, we can achieve comparable throughput to a fully buffered crossbar when the unbalanced coefficient ω is

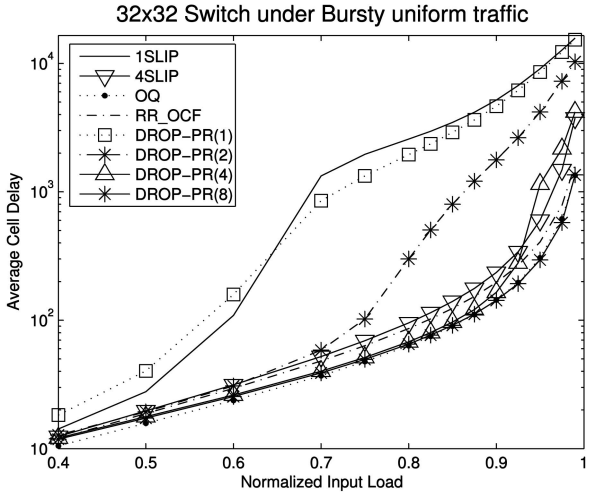

Fig. 15. Performance under bursty uniform arrivals.

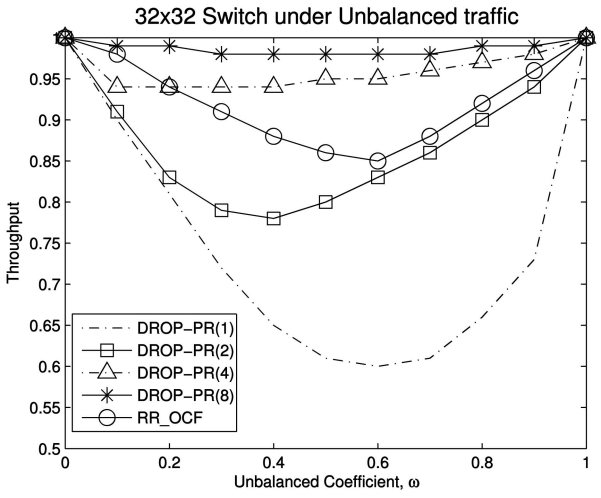

Fig. 17. Throughput performance under unbalanced traffic.

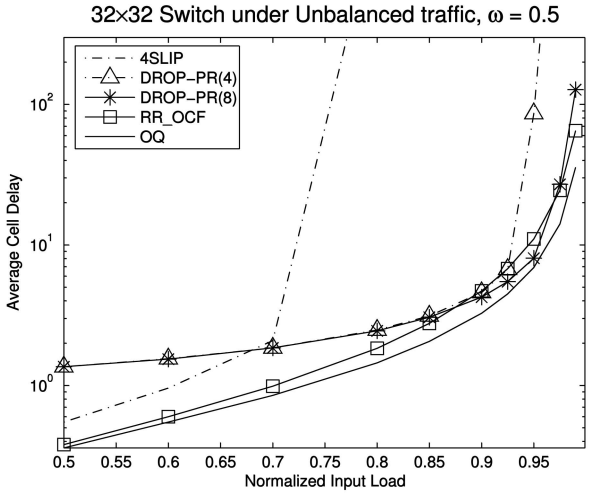

Fig. 18. Delay performance under unbalanced traffic, $\omega = 0.5$.

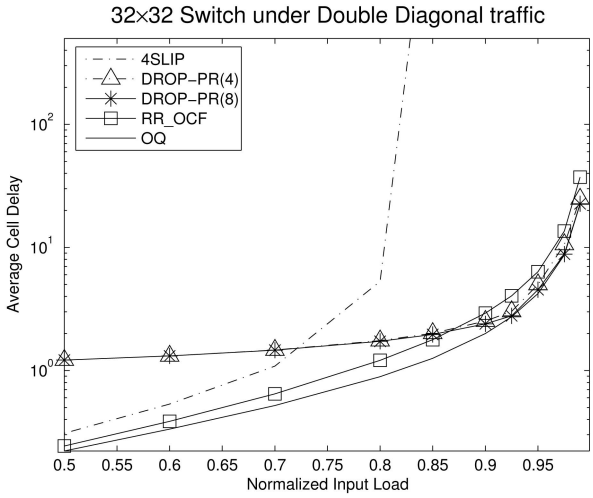

Fig. 19. Delay performance under double diagonal traffic.

higher than 0.6. This translates to a saving worth of up to 960 internal buffers. Setting $B = 4$, we can achieve higher throughput than a fully buffered crossbar. The ideal throughput of the PBC is reached when using eight internal buffers per output.

From Fig. 17, we can see that setting ω to be 0.5 corresponds to the pattern with the lowest performance (the hardest to schedule). We investigated the average cell delay of the DROP_PR algorithm iSLIP, RR_OCF, and OQ at that point. We set the unbalanced coefficient $\omega = 0.5$ and observed the average cell delay, as depicted in Fig. 18. We can see from Fig. 18 that 4SLIP (iSLIP with four iterations) saturates at 78 percent throughput whereas its PBC counterpart, DROP_PR(4), reaches a throughput of 96 percent. Again, when $B = 8$, DROP_PR(8) achieves full throughput and comparable delay to a full buffered crossbar (with $B = 32$ for each of the 32 outputs, amounting to 1,024 internal buffers as opposed to DROP_PR(8) that uses only 256 internal buffers) and an ideal OQ switching system. The same trend is observed under the double diagonal traffic pattern. We can see in Fig. 19 that 4SLIP cannot achieve more than 82 percent throughput, while both DROP_PR(4) and DROP_PR(8) achieve full throughput and better delay than RR_OCF that uses a fully buffered crossbar fabric. The good performance of DROP_PR(8) is attributable to its efficient use and balanced sharing of its limited number of internal buffers, in contrast to RR_OCF that uses excessive dedicated buffers per input/output pair of ports. We believe that more sophisticated PBC algorithms can emulate the optimal unbuffered scheduling algorithms, such as LQF and/or OCF [9] in a pipelined and distributed manner.

5 Summary

A novel PBC switching architecture is proposed in this paper. The PBC switch is designed to be the best compromise between unbuffered crossbars and fully buffered crossbars. On one hand, it overcomes the high cost of fully buffered crossbars that use N^2 internal buffers, by using a low number of internal buffers per output irrespective of N. On the other hand, it overcomes the scheduling complexity experienced by unbuffered crossbars by means of distributed and pipelined scheduling algorithms. We proposed a class of distributed and pipelined round-robin scheduling algorithms for the PBC architecture. In particular, the DROP-PR scheme was shown to have optimal performance under different traffic patterns and switch sizes.

The experimental results showed that a PBC switch with eight internal buffers per output exhibits ideal performance, irrespective of the switch size N. We also showed a design trade-off between using a traditional crossbar (whether fully buffered or unbuffered) and/or using the PBC switching system. This trade-off affords a switch designer wider performance and cost-based choices. We believe that the PBC architecture has good potential to become the architecture of choice for next generation routers. The reason for this is not only because the PBC achieves the best of both the unbuffered and fully buffered crossbars, but also because it can provide the opportunity to implement optimal bufferless scheduling algorithms in a pipelined and distributed fashion.

Acknowledgments

The author would like to thank Dimitris Theodoropoulos for his help with the hardware implementation in Section 3.5.

References

[1] N. McKeown, "A Fast Switched Backplane for a Gigabit Switched Router," *Business Comm. Rev.*, vol. 27, no. 12, 1997.
[2] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, "The Tiny Tera: A Packet Switch Core," *IEEE Micro*, vol. 17, no. 1, pp. 26-33, Jan./Feb. 1997.
[3] C. Minkenberg and T. Engbersen, "A Combined Input and Output Queued Packet-Switched System Based on a Prizma Switch-on-a-Chip Technology," *IEEE Comm. Magazine*, vol. 38, no. 2, pp. 70-77, Dec. 2000.
[4] F. Abel, C. Minkenberg, P. Luijten, M. Gusat, and I. Iliadis, "A Four-Terabit Packet Switch Supporting Long Round-Trip Times," *IEEE Micro*, vol. 23, no. 1, pp. 10-24, Jan./Feb. 2003.
[5] N. McKeown, "Scheduling Algorithms for Input-Queued Cell Switches," PhD thesis, Univ. of California at Berkeley, May 1995.
[6] N. McKeown, "iSLIP Scheduling Algorithm for Input-Queued Switches," *IEEE Trans. Networking*, vol. 7, no. 2, pp. 188-201, Apr. 1999.

[7] D.N. Serpanos and P.I. Antoniadis, "FIRM: A Class of Distributed Scheduling Algorithms for High-Speed ATM Switches with Input Queues," *Proc. IEEE INFOCOM,* Mar. 2000.
[8] Y. Jiang and M. Hamdi, "A Fully Desyncronized Round-Robin Matching Scheduler for a VOQ Packet Switch Architecture," *Proc. IEEE Workshop High Performance Switching and Routing (HPSR),* pp. 407-411, May 2001.
[9] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% Throughput in Input-Queued Switches," *IEEE Trans. Comm.,* vol. 47, no. 8, pp. 1260-1267, August 1999.
[10] A. Mekkittikul, "Scheduling Non-Uniform Traffic in High Speed Packet Switches and Routers," PhD thesis, Stanford Univ., Nov. 1998.
[11] S. Nojima, E. Tsutsui, H. Fukuda, and M. Hashimmoto, "Integrated Packet Network Using Bus Matrix," *IEEE Trans. Comm.,* vol. 5, no. 8, pp. 1284-1291, Oct. 1987.
[12] M. Nabeshima, "Performance Evaluation of Combined Input- and Crosspoint-Queued Switch," *IEICE Trans. Comm.,* vol. B83-B, no. 3, pp. 737-741, Mar. 2000.
[13] R. Rojas-Cessa, E. Oki, Z. Jing, and H.J. Chao, "CIXB-1: Combined Input One-Cell-Crosspoint Buffered Switch," *Proc. IEEE Workshop High Performance Switching and Routing (HPSR),* pp. 324-329, May 2001.
[14] L. Mhamdi and M. Hamdi, "MCBF: A High-Performance Scheduling Algorithm for Buffered Crossbar Switches," *IEEE Comm. Letters,* vol. 7, no. 9, pp. 451-453, Sept. 2003.
[15] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos, "Variable Packet Size Buffered Crossbar (CICQ) Switches," *Proc. IEEE Int'l Conf. Comm. (ICC),* pp. 1090-1096, June 2004.
[16] L. Mhamdi, C. Kachris, and S. Vassiliadis, "A Reconfigurable Hardware Based Embedded Scheduler for Buffered Crossbar Switches," *Proc. ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays (FPGA),* pp. 143-149, Feb. 2006.
[17] A.K. Choudhury and E.L. Hahne, "A New Buffer Management Scheme for Hierarchical Shared Memory Switches," *IEEE/ACM Trans. Networking,* vol. 5, no. 5, pp. 728-738, Oct. 1997.
[18] F.M. Chiussi and A. Francini, "A Distributed Scheduling Architecture for Scalable Packet Switches," *IEEE J. Selected Areas in Comm.,* vol. 18, no. 12, pp. 2665-2683, Dec. 2000.
[19] N. Chrysos and M. Katevenis, "Scheduling in Switches with Small Internal Buffers," *Proc. IEEE Global Comm. Conf. (Globecom),* pp. 614-619, Nov. 2005.
[20] N. Chrysos and M. Katevenis, "Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics," *Proc. IEEE INFOCOM,* Apr. 2006.
[21] S. Chuang, S. Iyer, and N. McKeown, "Practical Algorithms for Performance Guarantees in Buffered Crossbars," *Proc. IEEE INFOCOM,* Mar. 2005.
[22] J. Balfour and W.J. Dally, "Design Tradeoffs for Tiled CMP On-Chip Networks," *Proc. Int'l Conf. Supercomputing (ICS),* pp. 187-198, 2006.
[23] R.R. Cessa, "Design and Analysis of Reliable High-Performance Packet Switches," PhD thesis, Polytechnic Univ., Apr. 2001.
[24] S.T. Chuang, "Providing Performance Guarantees with Crossbar-Based Routers," PhD thesis, Stanford Univ., Dec. 2004.
[25] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High Speed Switch Scheduling for Local Area Networks," *ACM Trans. Computer Systems,* vol. 11, no. 4, pp. 319-352, Nov. 1993.
[26] R. Rojas-Cessa and Z. Dong, "Combined Input-Crosspoint Buffered Packet Switch with Flexible Access to Crosspoint Buffers," *Proc. IEEE Int'l Caribbean Conf. Devices, Circuits and Systems (ICCDCS),* Apr. 2006.
[27] P. Krishna, N.S. Patel, A. Charny, and R.J. Simcoe, "On the Speedup Required for Work-Conserving Crossbar Switches," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1528-1537, June 1999.
[28] K. Yoshigoe, K. Christensen, and A. Jacob, "The RR/RR CICQ Switch: Hardware Design for 10-Gbps Link Speed," *Proc. IEEE Int'l Performance Computing and Comm. Conf. (IPCCC),* pp. 481-485, Apr. 2003.
[29] P. Gupta and N. McKeown, "Design and Implementation of a Fast Crossbar Scheduler," *IEEE Micro,* vol. 19, no. 1, pp. 20-28, Jan./Feb. 1999.
[30] P. Giaccone, D. Shah, and B. Prabhakar, "An Implementable Parallel Scheduler for Input-Queued Switches," *IEEE Micro,* vol. 19, no. 1, pp. 1090-1096, Jan./Feb. 1999.

Lotfi Mhamdi received the Master of Philosophy (MPhil) degree in computer science from the Hong Kong University of Science and Technology (HKUST) in 2002 and the PhD degree in computer engineering from Delft University of Technology (TU Delft), The Netherlands, in 2007. He is currently with the Computer Engineering Laboratory, TU Delft. His research work spans the area of high-speed networks including the design, analysis, scheduling, and management of high-performance switches and Internet routers. He is/was a technical program committee member in various conferences, including the IEEE International Conference on Communications (ICC), the IEEE Workshop on High Performance Switching and Routing (HPSR), and the ACM/IEEE International Symposium on Networks-on-Chip (NoCS). He served as the publicity chair of the International Conference on Design and Technology of Integrated Systems in nanoscale era (DTIS). He is a member of the IEEE.