Chapter 5: Analyzing Buffered CIOQ Routers with Localized Memories

Feb 2008, Santa Cruz, CA

Contents

List of Dependencies

- **Background:** The memory access time problem for routers is described in Chapter 1. Section 1.5.2 describes the use of load balancing techniques, and Section 1.5.3 describes the use of caching techniques; both these techniques are used to alleviate the memory access time problems for the buffered CIOQ router.

Additional Readings

- **Related Chapters:** The load balancing technique to analyze the buffered combined input output queued (CIOQ) router, introduced in this chapter, is described in Section 4.9. The load balancing technique is also used to analyze the CIOQ router architecture in Chapter 4.

Table: *List of Symbols.*

B_{ij}	Crosspoint for Input i, Output j
c, C	Cell
DT	Departure Time
N	Number of Ports of a Router
R	Line Rate
S	Speedup
$SVOQ$	Super VOQ
T	Time Slot

Table: *List of Abbreviations.*

ASIC	Application Specific Integrated Circuit
CIOQ	Combined Input Output Queued Router
eDRAM	Embedded Dynamic Random Access Memory
FCFS	First Come First Serve (Same as FIFO)
OQ	Output Queued
PIFO	Push In First Out
SB	Single-buffered
QoS	Quality of Service
VOQ	Virtual Output Queue
WFQ	Weighted Fair Queueing

"A (stable) marriage can be ruined by a good memory".

— Anonymous[†]

5

Analyzing Buffered CIOQ Routers with Localized Memories

5.1 Introduction

In Chapter 4, we introduced the combined input output queued (CIOQ) router. CIOQ routers are widely used, partly because they have localized buffers that enable these routers to be easily upgraded. However they are hard to scale because the scheduler in a crossbar-based CIOQ router implements complex stable marriage algorithms [63]. This chapter is motivated by the following question: *Can we simplify the scheduler to make CIOQ routers more scalable?* The main idea is as follows —

> **Idea.** *"By introducing a small amount of buffering (similar to a cache) in the crossbar, we can make the scheduler's job much simpler. The intuition is that when a packet is switched, it can wait in the buffer; it doesn't have to wait until both the input and output are free at the same time".*

In other words, our scheduler doesn't have to resolve two constraints at the same time. As we will see, this will reduce the complexity of the scheduler from $O(N^2)$ to

[†]The exact version of the quote differs in literature.

$O(N)$ and remove the need for a centralized scheduler. In this chapter, we'll prove that anything we can do with a CIOQ router, we can do simpler with a buffered CIOQ router.

✎**Example 5.1.** Figure 5.1 shows a cross-sectional view of the crossbar ASIC in a CIOQ router. As can be seen, the central crossbar has to interface with all the line cards in the router. The current technology constraints are such that the size of the crossbar ASIC is determined by the number of interconnect pins. This leaves a large amount of on-chip area unused in the crossbar ASIC, which can be used to store on-chip buffers.

5.1.1 Goal

Our goal is to make high-speed CIOQ routers practical to build, while still providing deterministic performance guarantees. So, in what follows, we will not use complex *stable marriage algorithms* [63]. Instead we consider a crossbar with buffers (henceforth referred to as a "buffered crossbar") and answer the question —*What are the conditions under which a buffered crossbar can emulate an OQ router that supports qualities of service?*

In what follows, we'll show simple schedulers that make a buffered crossbar give deterministic performance guarantees – and we'll derive the conditions under which they are work-conserving and can emulate an OQ router that supports qualities of service.

5.1.2 Intuition

Researchers first noticed via simulation that the introduction of buffers to a crossbar can provide good statistical guarantees. They showed that the buffered crossbar can achieve high throughput for admissible uniform traffic with simple algorithms [99, 100, 101, 102]. Simulations also indicated that with a modest speedup, a buffered crossbar can closely approximate fair queueing [103, 104]. This confirms our intuition that the addition of a buffer can greatly enhance the performance of a crossbar-based fabric.

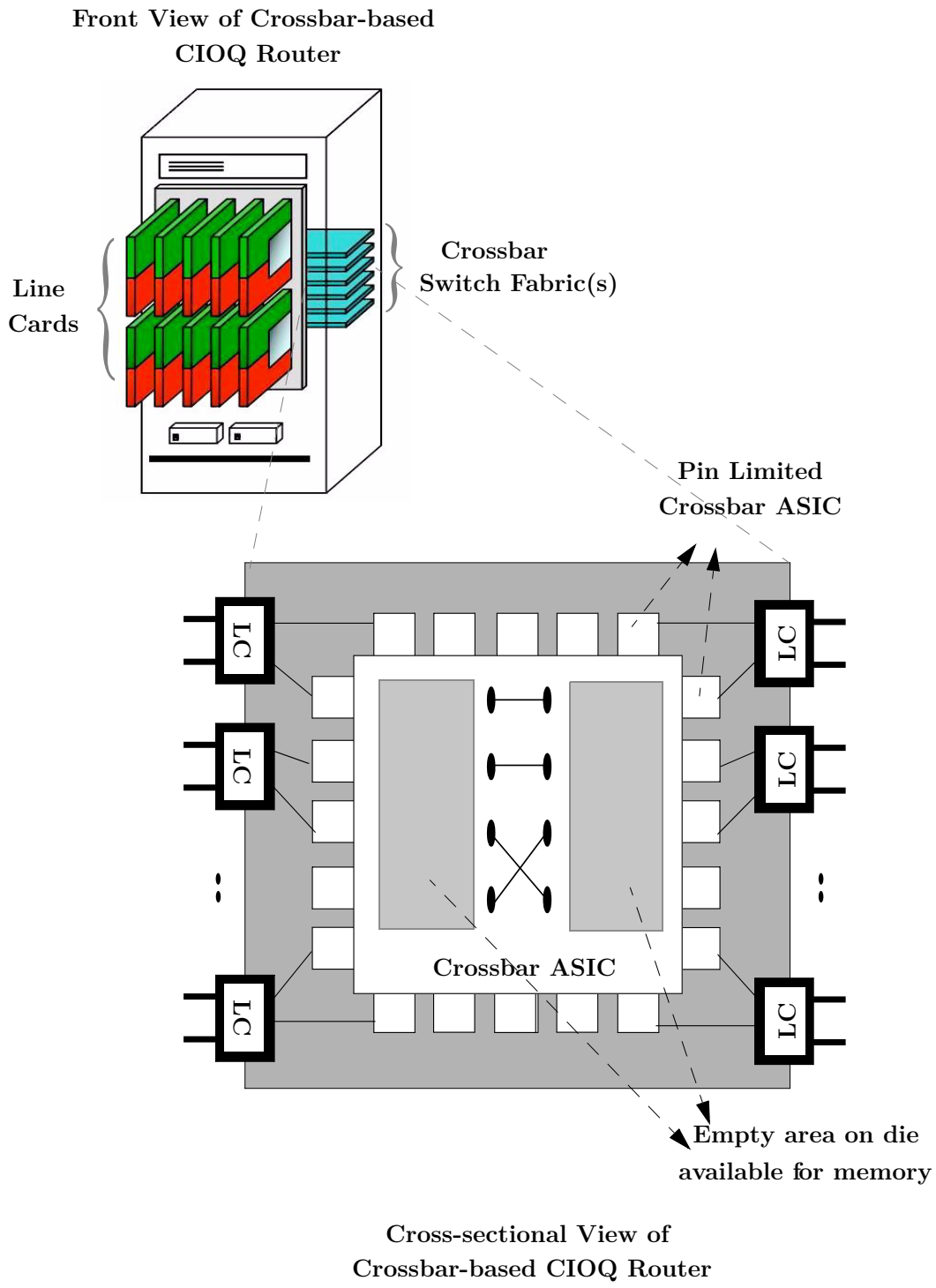

Figure 5.1: *Cross-sectional view of crossbar fabric.*

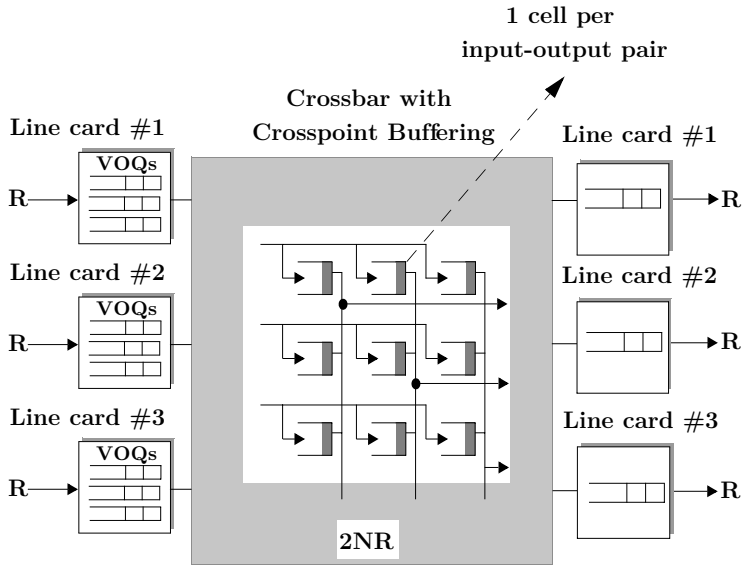

Figure 5.2: *The architecture of a buffered crossbar with crosspoint buffer.*

However, until recently there were no analytical results on guaranteed throughput to explain or confirm the observations made by simulations. The first analytical results were by Javidi *et al.* who proved that, with uniform traffic, a buffered crossbar can achieve 100% throughput [105]. More recently, Magill *et al.* proved that a buffered crossbar with a speedup of two can emulate an FCFS-OQ router [42, 106]. Magill *et al.* also showed that a buffered crossbar with k cells per crosspoint can emulate an FCFS-OQ router with k strict priorities.

5.2 Architecture of the Buffered CIOQ Router

Figure 5.2 shows a 3×3 buffered crossbar with line rate R. Similar to the crossbar router, arriving packets are buffered locally on the same line card on which they arrive, and held there temporarily. Later they are switched to the corresponding output line card, where they are again held temporarily before they finally leave the router. Fixed-length packets or cells wait in the VOQs to be transferred across the switch. Each crosspoint contains a buffer that can cache one cell. The buffer between input i

and output j is denoted as B_{ij}. When the buffer caches a cell, $B_{ij} = 1$, else $B_{ij} = 0$. In a sense, when the buffer is non-empty, it contains the head of a VOQ, and thus allows the output to *peek* into the state of the VOQ. Similar to the crossbar-based CIOQ router in the previous chapter, the inputs maintain an input priority queue (refer to Figure 4.6) to maintain the priority of cells. As in the previous chapter, there are two choices for implementation — (1) Keep arriving cells in VOQs, and a separate input priority queue to describe the order between the cells, or (2) Keep the cells sorted directly in an input priority queue.

The key to creating a scheduling algorithm is determining the input and output scheduling policy that decides how input and output schedulers pick cells. We will see that different policies lead to different scheduling algorithms.

5.2.1 Why Are Buffered Crossbars Practical?

Unlike a traditional crossbar, the addition of memory (which behaves like a cache) in a buffered crossbar, obviates the need for the computation of complex stable matching algorithms. This is because the inputs and outputs are no longer restricted to performing a matching in every time slot, and have more more flexibility as described below.

The scheduler for a buffered crossbar consists of $2N$ parts: N input schedulers and N output schedulers. The input schedulers (independently and in parallel) pick a cell from their inputs to be placed into an empty crosspoint. The output schedulers (independently and in parallel) pick a cell from a non-empty crosspoint destined for that output.

☞**Observation 5.1.** A traditional crossbar fabric is limited to performing no more than $N!$ permutations to match inputs to outputs. In contrast (depending on the occupancy of the crosspoints), the buffered crossbar can allow up to N^N matchings between inputs and outputs.

The N input and N output schedulers can be distributed to run on each input and output independently, eliminating the single centralized scheduler. They can be

pipelined to run at high speeds, making buffered crossbars appealing for high-speed routers.

☞**Observation 5.2.** It is interesting to compare this scheduler with the scheduler for an unbuffered crossbar-based CIOQ router described in Chapter 4. The scheduler in Chapter 4 also gave deterministic performance guarantees, but in order to provide these guarantees, our scheduling algorithm required the inputs and outputs to keep a list of preferences and convey these preferences to a scheduler. This meant that the scheduler had to be aware of the states of all inputs and outputs, and be centrally located. The scheduler computed a matching by running a version of the *stable marriage algorithm* [63]. In [13], the authors describe two scheduling algorithms that can compute stable matching, the faster of which still takes $O(N)$ iterations to compute a matching. The high communication overhead between the inputs and outputs, the amount of state that needs to be maintained and communicated to the centralized scheduler, and the implementation complexity, make stable matching algorithms hard to scale when designing high-speed routers.

Of course, simplifying the scheduler requires a more complicated crossbar capable of holding and maintaining N^2 packet buffers on-chip. In the past, this would have been prohibitively complex: the number of ports and the capacity of a crossbar switch were formerly limited by the N^2 crosspoints that dominated the chip area; hence the development of multi-stage switch fabrics, *e.g.*, the Clos, Banyan, and Omega switches, based on smaller crossbar elements. Now, however, crossbar switches are limited by the number of pins required to get data on and off the chip [107].

✎**Example 5.2.** Improvements in process technology, and reductions in geometries, mean that the logic required for N^2 crosspoints is small compared to the chip size required for N inputs and N outputs. The chips are pad-limited, with an under-utilized die. A buffered crossbar can use the

unused die for buffers, and further improvements in on-chip eDRAM memory technology [26] allow for storing a large amount of memory on-chip. With upcoming 45nm chip technology, ~128Mb of eDRAM can easily fit on an average chip. This can easily support a 256×256 router (with 256-byte cells), making a buffered crossbar a practical proposition.[1]

5.3 Applying the Pigeonhole Principle to Analyze Buffered Crossbars

In Chapter 4, we introduced the extended pigeonhole principle (Algorithm 4.2) as a general technique to analyze whether a router can emulate an OQ router (and as a consequence give deterministic performance guarantees). The crossbar-based CIOQ router in Chapter 4 was able to emulate an OQ router because the stable matching algorithms that we described (and first mentioned in [13]) ensured that the pigeonhole principle is satisfied for every cell in every time slot.

5.3.1 Satisfying the Properties for the Extended Pigeonhole Principle

Since we are also interested in emulating an OQ router, we will choose a scheduling algorithm that also attempts to satisfy the conditions of the pigeonhole principle. Our scheduling algorithm consists of two parts, one each for the input and the output scheduler. This is shown in Algorithm 5.1. To show that our scheduling algorithm meets the pigeonhole principle, recall the two properties that we introduced in Section 4.9:

✓**Property 4.1.** **Induction Step:** In every time slot, the switch scheduler does the following at least *twice* — it either decrements the size of the contention

[1] Some detailed considerations regarding the implementation of buffered crossbars are described in [108].

Algorithm 5.1: Buffered crossbar scheduler.

```
1 for each scheduling slot t do
2     Input Scheduler:
3         for each input i do
4             Serve the first cell (destined to any output) that it can find in the input
              priority queue, whose crosspoint is empty, i.e., B_i* = 0.
5     Output Scheduler:
6         for each output j do
7             Serve the first cell with the earliest departure time, from any crosspoint
              B_*j which is not empty, i.e., B_*j = 1.
```

set or increments the size of the opportunity set for every cell waiting at the input, *i.e., it either decrements the number of pigeons or increments the number of pigeonholes for every cell.*

Consider a cell c in the input priority queue for input i destined to output j. If the cell is chosen by the input scheduler, then it is transferred to crosspoint B_{ij} and is available to the output for reading at any time. We no longer need to consider it. If the cell is not chosen, then consider the state of the crosspoint buffer, B_{ij}, at the beginning of every time slot:

1. **If $B_{ij} = 0$:** The input scheduler will select some other cell with a higher priority than cell c. So, *the input scheduler decrements the size of the contention set (pigeons) for cell c.*
2. **If $B_{ij} = 1$:** The output scheduler will select either the cell in the crosspoint buffer B_{ij} (which has an earlier departure time than cell c[2]), or some other cell with an even earlier departure time than the cell in crosspoint B_{ij}. In either case, *the output scheduler increments the number of opportunities*[3] *(pigeonholes)*

[2]This implicitly assumes that the cell in the crosspoint buffer B_{ij} has an earlier departure time than all cells in VOQ_{ij}. How this is ensured is explained in the later sections.

[3]Note that the number of opportunities for a cell is simply the number of cells in the output with an earlier departure time.

for cell c.

So, for every cell c that remains at the input at the end of a time slot, either the input scheduler reduces the number of contending cells (pigeons), or the output scheduler increases the number of opportunities (pigeonholes).[4] If the speedup of the crossbar is $S = 2$, then similar to Section 4.9, this is enough to offset the potential increment of the contention set (pigeonholes) and the definite decrement of the opportunity set (pigeons) in every time slot due to arriving and departing cells.

✓**Property 4.2.** **Induction Basis Case:** On arrival, a cell is inserted into a position in the input priority queue such that it has equal or more opportunities to leave than the number of contentions, *i.e., it has equal or more pigeonholes than the number of pigeons.*

We will make our cell insertion policy identical to that introduced in Section 4.9, and so it satisfies Property 4.2.

5.4 Analyzing Buffered FCFS-CIOQ Routers Using the Extended Pigeonhole Principle

In the previous section, in order to meet the Property 4.1 we had to assume that the cell in the crosspoint B_{ij} has an earlier departure time than the cells in VOQ_{ij}. In the case of an FCFS router this is trivially true – for cells that belong to the same VOQ, the input scheduler will only transfer the cell with the earliest departure time (since it will have a higher priority than other cells in the same VOQ). Also, cells that arrive later for the same VOQ will have a later departure time than the cell in the crosspoint, because of the FCFS nature of the router. It follows that all the conditions to meet the extended pigeonhole principle (Algorithm 4.2) are satisfied. Similar to Section 4.9, we can state that a buffered CIOQ router can emulate an FCFS-OQ router. These observations result in the following theorem:

[4]This is in contrast to stable marriage algorithms, which try to resolve both the input and output contentions at once.

Theorem 5.1. *(Sufficiency, by Citation) A buffered crossbar can emulate an FCFS-OQ router with a crossbar bandwidth of $2NR$ and a memory bandwidth of $6NR$.*

Proof. This was first proved by Magill et al. in [42, 106]. ❐

✂Box 5.1: Work Conservation without Emulation✂

We can ask an identical question to the one posed in the previous chapter: What are the conditions under which a buffered CIOQ router is work-conserving *without* mandating FCFS-OQ emulation?

Recall that for a work-conserving router, we only need to ensure that its outputs are kept busy. So, the output scheduler for a work-conserving buffered crossbar can choose to serve *any* non-empty crosspoint, thus ignoring the order of cells destined to an output. This simplified scheduling policy would still meet the requirements of the modified pigeonhole principle for work-conserving routers (as described in Algorithm 4.3). This leads to the following theorem:

Theorem 5.2. *A buffered crossbar (with a simplified output scheduler) is work-conserving with a crossbar bandwidth of $2NR$ and a memory bandwidth of $6NR$.*

5.5 Crosspoint Blocking

In the previous section, we simplified Magill's work on the FCFS buffered crossbar using the pigeonhole principle. We now extend this work and consider WFQ routers. When a cell arrives in a WFQ router, the scheduler picks its departure order relative to other cells already in the router. If the cell has a very high priority, it could, for example, be scheduled to depart immediately, ahead of all currently queued cells. This will cause problems for the buffered crossbar. Imagine the situation where the crosspoint buffer is non-empty and a new cell arrives that needs to leave *before* the cell in the crosspoint buffer. Because there is no way for the new cell to overtake the cell in the crosspoint buffer, we say there is *"crosspoint blocking"*.

Crosspoint blocking is bad, because the cell in the crosspoint B_{ij} may not have an earlier departure time than the cells in VOQ_{ij}. So we cannot satisfy Property 4.1 or

meet the requirements of the extended pigeonhole principle. This is a problem unique to supporting WFQ policies on a buffered crossbar, and did not occur for the FCFS policy that we considered in the previous section.[5]

How can we overcome crosspoint blocking? There are three ways to alleviate this problem — (1) fix it, (2) work around it, or (3) prevent it from occurring. We will consider each of them below.

✍**Method 1.** **What if we can recall less-urgent cells from the crosspoint?** We can fix the crosspoint blocking problem by allowing the input to replace the cell in the crosspoint buffer with a more urgent cell, if we swap out the old one and exchange it for the new one. The new cell is put into position in the crosspoint buffer, ready to be read by the output scheduler in time to leave the switch before its deadline. Logically, the cell that was previously in the crosspoint buffer is recalled to the input, where it is treated like a newly arriving cell.

✍**Method 2.** **What if we can bypass less-urgent cells in the crosspoint?** We can work around crosspoint blocking by allowing the more urgent arriving cell to bypass the cell in the crosspoint buffer. The inputs could continuously inform the outputs when more urgent cells than those in the crosspoint buffer have arrived. In a sense, the output can choose to ignore the crosspoint buffer and directly read the more urgent cells from the input. This means that the outputs treat the buffered crossbar similar to a traditional crossbar.

✍**Method 3.** **What if we can disallow less-urgent cells from entering the crosspoint?** We can prevent the crosspoint blocking problem if the cells that cause crosspoint blocking are never put there in the first place! A way to do this is to ensure that a cell is sent to the crosspoint only if we know that the output will read it immediately.

[5]In contrast, the analysis of FCFS and PIFO crossbar CIOQ routers in Theorem 4.3 was identical, because crosspoint blocking does not happen in unbuffered crossbars.

5.6 Analyzing Buffered PIFO CIOQ Routers Using the Extended Pigeonhole Principle

We will now derive three results to support WFQ-like policies on the buffered crossbar, based on the three unique methods described above.

5.6.1 Emulating a PIFO-OQ Router by Recalling Cells

We will need extra speedup in the crossbar, to allow time for the old (less-urgent) cell in the crosspoint to be replaced in the crosspoint buffer by the new cell. To perform this swapping operation, the switch now has a speedup of three. We don't need extra crossbar speedup to retrieve the old (less-urgent) cell from the crossbar and send it back to the input — in practice, the input would keep a copy of cells currently in the crosspoint buffer, and would simply overwrite the old one. The memory bandwidth on the input also does not need to change, since the old cell that is retrieved from the crossbar does not need to be re-written to the input VOQ. This leads us to the following theorem:

Theorem 5.3. *(Sufficiency, by Reference) A buffered crossbar can emulate a PIFO-OQ router with a crossbar bandwidth of* $3NR$ *and a memory bandwidth of* $6NR$.

Proof. This follows from the arguments above and 5.1. The original proof (which does not use the pigeonhole principle) appears in our paper [43]. ❐

5.6.2 Emulating a PIFO-OQ Router by Bypassing Cells

In order to bypass cells in the crosspoint, we needed to continually inform the outputs about newly arriving cells. But this creates a scheduling problem: it means we no longer split the scheduling into independent input and output stages, since the outputs must wait for the inputs to communicate if there are more urgent arriving cells. So the scheduler will become similar to (and just as complex as) the scheduler in the

unbuffered crossbar in Chapter 4. Of course, since the buffered crossbar can perform any matching that a traditional crossbar can, we have the following obvious corollary:

Corollary 5.1. *(Sufficiency, by Citation) A crossbar CIOQ router can emulate a PIFO-OQ router with a crossbar bandwidth of* $2NR$ *and a total memory bandwidth of* $6NR$.

Proof. This follows from Theorem 4.3. ❐

5.6.3 Emulating a PIFO-OQ Router by Disallowing Less Urgent Cells

We are interested in a mechanism for knowing when an output will read a cell, so that the inputs can send only these cells into the crosspoint. How can the input know when an output will read a cell? One way of doing this is based on the following idea:

> **Idea.** *"Instead of storing the cell in the crosspoint buffer, we can store a "cell header or identifier" that contains the departure time. The output can still pick a cell according to the time it needs to leave. The actual cell can be transferred later, once the output has made its decision".*

This means we only send cells through the switch when they *really* need to be transferred. We don't need the extra speedup to overwrite cells in the crosspoints. We call this approach *header scheduling*.

The process of scheduling and transferring cells would now take place in two distinct phases, just as in an unbuffered crossbar. First, scheduling would be done by inputs and outputs: the inputs would send cell identifiers to their corresponding crosspoints, and the outputs would pick, or grant to, a cell identifier in the crosspoint. In the second phase, the actual cells would be transferred according to the grants made by the outputs. But we are not quite done:

☞**Observation 5.3.** Since the N output schedulers all operate independently, they could temporarily choose cell identifiers that are destined to them, from the same input. Since the actual cells are *not* as yet in the crosspoint buffer, they have to be provided by the inputs. Since the cell identifier grants come to the inputs in a bursty manner, the actual cells also reach the crosspoints in a bursty manner. It can be shown that the outputs need no more than N cells of buffering to accommodate this burst (see Appendix E).

There are two ways to accommodate this burst, both of which come at an expense.

1. **Expand the size of crosspoints:** We could modify the crosspoint buffer B_{ij}, to store up to N cells. This results in a cache size of N^3 cells in the crossbar as a whole. While this will require much more storage, it might make sense for small values of N. Our buffered crossbar will be similar to that shown in Figure 5.2, except that each crosspoint can store N cells.

2. **Share the crosspoints destined to an output:** We could take advantage of the fact that the burst size per output (irrespective of the inputs) is bounded by N cells, and share the crosspoint buffer for a specific output. Instead of one buffer per crosspoint buffer, there will now be N buffers per output as shown in Figure 5.4. The total cache size is still N^2 cells in the crossbar, but the buffers are dedicated to outputs, rather than input/output pairs. This requires us to modify the design of the buffered crossbar. Also note that in such a design, the per-output buffer must have the memory access rate to allow up to N cells to arrive simultaneously.

The above modifications to the crossbar allow us to emulate a PIFO-OQ router and provide delay guarantees without requiring any additional crossbar speedup or memory bandwidth.

✂Box 5.2: A Digression to Randomized Algorithms✂

While randomized algorithms are not a focus of this thesis, the buffered crossbar is a good example to show their power. Consider the scheduler in Algorithm 5.2. The input and output schedulers pick a cell to serve independently and at random. We can prove the following:

Theorem 5.4. *A buffered crossbar with randomized scheduling, can achieve 100% throughput with a crossbar bandwidth of* $2NR$ *and a memory bandwidth of* $6NR$.

Algorithm 5.2: A randomized scheduler.

```
1 for each scheduling slot t do
2     for each input i do
3         Serve any non-empty VOQ for which B_{i,*} = 0.
4     for each output j do
5         Serve any crosspoint buffer for which B_{*,j} = 1.
```

What follows is an intuition of the proof.[a] We define a *super queue*, $SVOQ_{ij}$ to track the evolution of each VOQ_{ij}. Let $SVOQ_{ij}$ denote the sum of the cells waiting at input i, and the cells destined to output j (including cells in the crosspoint for output j), as shown in Figure 5.3:

$$SVOQ_{ij} = \sum_k VOQ_{ik} + \sum_k (VOQ_{kj} + B_{kj}). \qquad (5.1)$$

First we show that the expected value of every super queue is bounded. It is easy to see, when VOQ_{ij} is non-empty, that $SVOQ_{ij}$ decreases in every scheduling opportunity. There are two cases:

Case 1: $B_{ij} = 1$. Output j will receive some cell; $\sum_k (VOQ_{kj} + B_{kj})$ decreases by one.

Case 2: $B_{ij} = 0$. Input i will send some cell; $\sum_k VOQ_{ik}$ decreases by one.

With $S = 2$, $SVOQ_{ij}$ will decrease by two per time slot. When the inputs and outputs are not oversubscribed, the expected increase in $SVOQ_{ij}$ is strictly less than two per time slot. So the expected change in $SVOQ_{ij}$ is negative over the time slot; this means that the expected value of $SVOQ_{ij}$ is bounded. This in turn implies that the expected value of VOQ_{ij} is bounded and the buffered crossbar has 100% throughput.

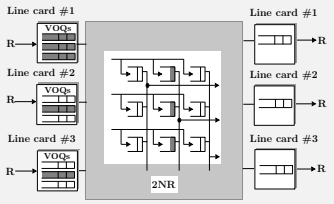

Figure 5.3: *SuperVOQ*$_{1,2}$

[a] The proof uses fluid models [12], and appears in Appendix D, and in our paper [43].

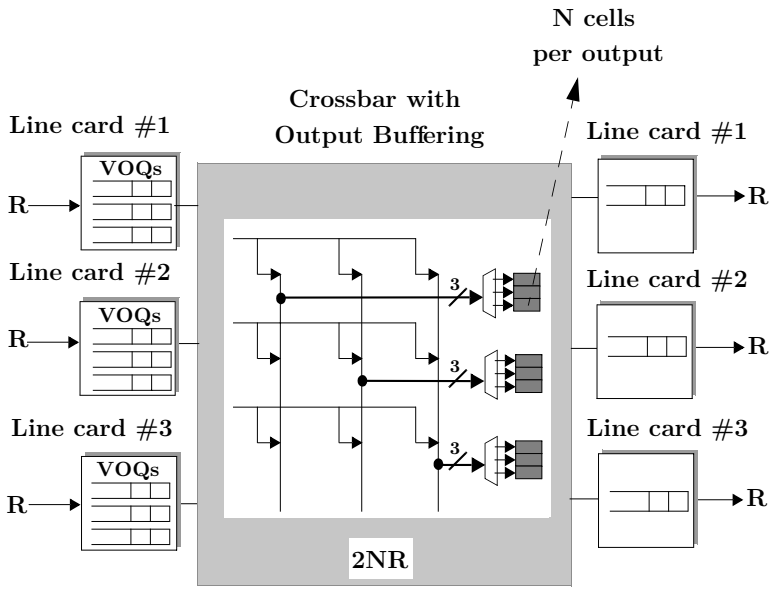

Figure 5.4: *The architecture of a buffered crossbar with output buffers.*

☞**Observation 5.4.** We note that due to the bursty nature of the arrival of cells into the crossbar, the cells can get delayed in reaching their outputs. The bursts of N cells take up to $N/2$ time slots (because the speedup is two) to reach their outputs. So the emulation is within a bound of $N/2$ time slots.

Based on the above observations, we are now ready to prove the following theorem:

Theorem 5.5. *(Sufficiency, by Reference) A modified buffered crossbar can emulate a PIFO-OQ router with a crossbar bandwidth of* $2NR$ *and a memory bandwidth of* $6NR$.

Proof. A detailed proof is available in Section 6.C in our paper [43] and also in Appendix E. ❒

Table 5.1: *Comparison of emulation options for buffered crossbars.*

Buffered Xbar Architecture	Cache Size	Type	# of memories	Memory Access Rate	Total Memory BW	Switch BW	Comment
Crosspoints (Box 5.2)	N^2	VII.R	$2N$	$3R$	$6NR$	$2NR$	100% throughput with trivial input and output scheduler.
Crosspoints (Theorem 5.1)	N^2	VII.D	$2N$	$3R$	$6NR$	$2NR$	Emulates FCFS-OQ with simple input and output scheduler.
Crosspoints (Theorem 5.3)	N^2	VII.D	$2N$	$3R$	$6NR$	$3NR$	Emulates WFQ OQ with simple input and output scheduler.
Crosspoints (Corollary 5.1)	N^2	VII.D	$2N$	$3R$	$6NR$	$2NR$	Emulates WFQ OQ with complex stable marriage algorithm.
Crosspoints (Theorem 5.5)	N^3	VII.D	$2N$	$3R$	$6NR$	$2NR$	Emulates WFQ OQ with N buffers per input-output pair, simple header scheduler.
Output Buffers (Theorem 5.5)	N^2	VII.D	$2N$	$3R$	$6NR$	$2NR$	Emulates WFQ OQ with N buffers per output, simpler header scheduler.

5.7 Conclusions

We set out to answer a fundamental question about the nature of crossbar fabrics: Can the addition of buffers make the crossbar-based CIOQ router practical to build, yet give deterministic performance guarantees?

Our results show that this is possible. Although the buffered crossbar is more complex than the traditional crossbar, the crossbar speedup is still two, the memory bandwidth and pin count is the same as in a crossbar-based CIOQ router, and no memory needs to run faster than twice the line rate. The scheduling algorithm is simple, and has the nice property of two separate, independent phases to schedule inputs and outputs, allowing high-speed, pipelined designs. Also, the use of the extended pigeonhole principle has allowed us a better understanding of the operation of the buffered crossbar, and we were able to develop a number of practical solutions (summarized in Table 5.1) that achieve our goal.

The use of buffered crossbars in some of Cisco's high-speed Ethernet switches and Enterprise routers [4] (though not with the exact architecture and algorithms suggested here) attests to the practicality of the overall architecture. We also currently plan to deploy buffered crossbars in the aggregated campus router [109] market, where the crossbar speeds exceed 400 Gb/s. Until crossbar switches and schedulers

improve in speed, we believe that the buffered crossbar provides a quick and easy evolutionary path to alleviate the problems inherent in a crossbar, and give statistical and deterministic performance guarantees.

Summary

1. CIOQ routers are widely used. However, they are hard to scale (while simultaneously providing deterministic guarantees) because of the complexity and centralized nature of the scheduler. In this chapter, we explore how to simplify the scheduler.
2. We show that by introducing a small amount of buffering in the crossbar (similar to a cache), we can make the scheduler's job much simpler. We call these "buffered crossbar" routers.
3. In a buffered crossbar, each input i and output j has a dedicated buffer in the crossbar. We call these "crosspoint buffers" (denoted B_{ij}), and a buffered crossbar has a total of N^2 crosspoint buffers.
4. The intuition is that when a packet is switched, it doesn't have to wait until both the input and output are both free at the same time. In other words, the scheduler doesn't have to resolve two constraints simultaneously. This can reduce the complexity of the scheduler from $\Theta(N^2)$ for crossbar-based CIOQ routers (see Chapter 4) to $\Theta(N)$ for buffered crossbars.
5. In this chapter, we prove that anything we can do with a CIOQ crossbar-based router, we can do more simply with a buffered crossbar.
6. The scheduler for a buffered crossbar consists of $2N$ parts: N input schedulers and N output schedulers. The schedulers can be distributed to run on each input and output independently, eliminating the need for a single centralized scheduler. They can be pipelined to run at high speeds, making buffered crossbars appealing for high-speed routers.
7. We prove that a buffered crossbar router with a simple scheduler that is distributed and runs in parallel can achieve 100% throughput (Theorem 5.4) and is work-conserving (Theorem 5.2), with a crossbar bandwidth of $2NR$ and a memory bandwidth of $6NR$.
8. In [42, 106] Magill et al. proved that a buffered crossbar can emulate an FCFS-OQ router with a crossbar bandwidth of $2NR$ and a memory bandwidth of $6NR$ (Theorem 5.1).
9. Unfortunately, we cannot extend the results in [42, 106] for a router that supports qualities of service. This is because buffered crossbars suffer from a type of blocking called

"crosspoint blocking".

10. Crosspoint blocking occurs when a lower-priority cell from some input i destined to some output j resides in the crosspoint B_{ij}. This cell prevents a higher-priority cell from the same input-output pair from being transferred to the crossbar.
11. We introduce three methods to eliminate crosspoint blocking. These methods trade off complexity of implementation for additional crossbar bandwidth. In some cases, we have to increase the amount of buffering (or the organization of the buffers) in the crossbar to overcome crosspoint blocking.
12. We show that a buffered crossbar can emulate an OQ router that supports qualities of service with a crossbar bandwidth of $3NR$ and a memory bandwidth of $6NR$ (Theorem 5.3).
13. We also show that a modified buffered crossbar can emulate an OQ router that supports qualities of service with a crossbar bandwidth of $2NR$ and a memory bandwidth of $6NR$ (Theorem 5.5).
14. There are two options to build a modified buffer crossbar. In one option, we can have N buffers per crosspoint (where each crosspoint is dedicated, as before, to an input-output pair), for a total of N^3 buffers.
15. Another option is to have N buffers dedicated to each output, for a total of N^2 buffers in the crossbar.
16. Our results are practical, and we believe that the buffered crossbar provides a quick and easy evolutionary path to alleviate the problems inherent in a crossbar and give statistical and deterministic performance guarantees.