2009-80

Design and Evaluation of a Practical, High Performance Crossbar Scheduler

Authors: Jonathan Turner

Corresponding Author: jon.turner@wustl.edu

Web Page: http://arl.wustl.edu/~jst

Abstract: The Least Occupied Output First (LOOFA) scheduler is one of several unbuffered crossbar schedulers that provides strong performance guarantees when operated with a speedup of 2 or more. Because LOOFA requires the computation of a maximal matching, it has been considered too slow for use in systems with link rates of 10 Gb/s or more. This paper studies an approximate variant of LOOFA described briefly in [16]. We introduce a general family of schedulers that allows for partial sorting and that includes the LOOFA scheduler as a special case. We show that all schedulers in this class are work-conserving and use this to provide insight into the operation of the Approximate LOOFA scheduler and a stronger motivation for its use. We provide a detailed design of the ALOOFA scheduler in order to evaluate its implementation complexity and performance characteristics. We also introduce a simple, natural lower bound on the performance of crossbar schedulers and use it to show that a previously proposed “stress test” traffic pattern is in fact difficult to schedule well. Our result implies that non-trivial speedups are required for ideal worst-case scheduling performance, something that has been generally assumed to be true, but never conclusively demonstrated. We also compare the performance of both LOOFA variants to our lower bound on stress test traffic and observe that for speedups between 1 and 2, the performance of both variants stays within 25% of the lower bound, and that the performance characteristics

Type of Report: Other

Design and Evaluation of a Practical, High Performance Crossbar Scheduler

Jonathan Turner
jon.turner@wustl.edu

Abstract

The Least Occupied Output First (LOOFA) scheduler is one of several unbuffered crossbar schedulers that provides strong performance guarantees when operated with a speedup of 2 or more. Because LOOFA requires the computation of a maximal matching, it has been considered too slow for use in systems with link rates of 10 Gb/s or more. This paper studies an approximate variant of LOOFA described briefly in [16]. We introduce a general family of schedulers that allows for partial sorting and that includes the LOOFA scheduler as a special case. We show that all schedulers in this class are work-conserving and use this to provide insight into the operation of the Approximate LOOFA scheduler and a stronger motivation for its use. We provide a detailed design of the ALOOFA scheduler in order to evaluate its implementation complexity and performance characteristics. We also introduce a simple, natural lower bound on the performance of crossbar schedulers and use it to show that a previously proposed "stress test" traffic pattern is in fact difficult to schedule well. Our result implies that non-trivial speedups are required for ideal worst-case scheduling performance, something that has been generally assumed to be true, but never conclusively demonstrated. We also compare the performance of both LOOFA variants to our lower bound on stress test traffic and observe that for speedups between 1 and 2, the performance of both variants stays within 25% of the lower bound, and that the performance characteristics of the two variants are essentially indistinguishable.

1. INTRODUCTION

The Least Occupied Output First (LOOFA) crossbar scheduling algorithm [6] is one of a number of such algorithms that offers strong performance guarantees [2,4,5,10,17,20]. In particular, when used with crossbars with a speedup of 2 or more, it guarantees work-conserving behavior, ensuring that none of the output link capacity is lost due to less-than-ideal scheduling. The LOOFA scheduler also provides better fairness properties than other schedulers with similar worst-case performance, when operated with smaller speedups. Unfortunately, schedulers for unbuffered crossbars with strong performance guarantees have yet to see much practical application, as they are difficult to implement in high performance systems. The key difficulty has been the requirement that the scheduler compute maximal matchings, or in some cases, stable matchings, in order to provide the performance guarantee.

The Approximate LOOFA (ALOOFA) scheduler was introduced briefly in [16], as a practical variant of the LOOFA scheduler. The authors argued that ALOOFA could be implemented efficiently in hardware and provided performance that was essentially indistinguishable from LOOFA. One of the intriguing aspects of the ALOOFA scheduler is that it does not compromise in the matching computation, but in the

sorting operation needed to maintain an ordered list of output queue lengths. The authors argue that since queue lengths change slowly, their order can be adequately maintained by applying a single odd-even sorting step after each operational cycle.

This paper provides a more detailed examination of the ALOOFA scheduler, in order to determine if ALOOFA is in fact a practical alternative for high performance crossbars. In order to gain greater insight into ALOOFA, we define a general class of scheduling algorithms that allows for partial sorting and show that all schedulers in this class are work-conserving when operated with speedups of 2. This class includes LOOFA, but not ALOOFA. Still, our analysis does provide a stronger justification for the use of partial sorting in ALOOFA and allows us to more clearly identify the important differences between these two scheduling algorithms. We describe a complete hardware implementation of ALOOFA in order to more fully evaluate its hardware complexity and performance. This reveals that while the essential assertion in [16] remains valid, the complexity and performance of the ALOOFA scheduler is determined largely by the circuitry needed to support various overhead functions, not the central matching operation. We also introduce a non-trivial lower bound on the performance of unbuffered crossbar schedulers and show that a so-called "stress test" traffic pattern introduced in [16] is in fact challenging, when it comes to worst-case performance. Specifically, we show that for this traffic pattern, no crossbar scheduler (including "offline" schedulers that have advance knowledge of the complete traffic pattern) can match the performance of an ideal output-queued switch unless operated with a non-trivial speedup. This confirms a widely believed, but previously unverified conjecture. Finally, we study the performance of both LOOFA and ALOOFA on stress test patterns and show that they typically complete the transfer of all cells through the crossbar in a time that is within 25% of that implied by the lower bound.

The performance of crossbar schedulers for packet switches has been studied extensively since the early 1990s. Anderson, et al described the *Parallel Iterative Matching* (PIM) method used in the DEC AN2 switch in 1993 [1]. This was the inspiration for a number of subsequent schedulers, including the popular *i*-SLIP method of McKeown [12] in 1999. The performance studies of this period were largely based on simulation, and typically focused on traffic patterns that were relatively benign in nature. The late nineties saw the appearance of the first worst-case results that provided strong performance guarantees, independent of the incident traffic. The simplest such results established conditions under which a crossbar scheduler was *work-conserving*, meaning that it never failed to forward a cell to an output, if there was a cell for that output, anywhere in the system [6]. Other papers established conditions under which a scheduler was not only work-conserving but forwarded cells in the order they were received, regardless of the input on which they arrived [17]. In a seminal paper, Chuang et al, showed that certain schedulers can exactly emulate an output-queued switch that uses any one of a wide class of queueing disciplines [3]. All of these results require that the crossbar be operated with a speedup of 2 relative to the external links, and require the computation of either a maximal matching or a stable matching. This requirement makes them difficult to implement in high performance systems and has limited their practical application.

More recently, there has been a growing interest in buffered crossbars that are capable of storing a small number of cells or packets at each of a crossbar's crosspoints [5,7,8,10,14,15,18]. While this significantly increases the circuit complexity of the crossbar, ongoing improvements in integrated circuit density make it a practical alternative. What makes buffered crossbars attractive is the fact that they are easier to control, making it possible to obtain good worst-case performance guarantees using relatively simple scheduling methods [3]. Buffered crossbars also make it possible for inputs and outputs to operate asynchronously, allowing direct switching of variable length packets. References [2,20] shows how worst-case performance results developed for cell-based switches can be extended to buffered crossbars that switch variable length packets.

There is a separate category of crossbar performance results that focuses on performance for random traffic that is *admissible*, meaning that the incoming traffic does not exceed the capacity of any outgoing link for an extended period of time [9,11,13]. Results of this form are often referred to as 100% through-

put results, or stability results. These results typically do not require a speedup, but also fail to provide the kind of strong guarantee that the worst-case results provide. Given the highly variable and unpredictable nature of internet traffic, schedulers that can provide worst-case guarantees seem preferable, so long as their cost is not exorbitant.

In Section 2, we introduce a family of crossbar schedulers that generalize the LOOFA scheduler and show that all schedulers in the class are work-conserving. Section 3, we provide a complete description of the ALOOFA scheduler and explain the basic characteristics that make it an attractive candidate for high performance crossbars. In Section 4, we describe a specific implementation of the ALOOFA scheduler, and use it to evaluate the circuit complexity and performance. In Section 5, we present bounds on the scheduling performance of crossbar schedulers and examine how LOOFA and ALOOFA perform, relative to these bounds. We conclude in Section 6 with some closing remarks.

2. A FAMILY OF WORK-CONSERVING CROSSBAR SCHEDULERS

We are concerned with scheduling the transfer of fixed length cells from inputs to outputs of a crossbar switch with no internal buffering. To enable good worst-case performance, crossbars can be operated with a *speedup S* that supports a peak cell transfer rate through the crossbar that is S times larger than the rate at which cells arrive on the inputs. For typical values of S, this means that queueing is required at both inputs and outputs to the crossbar. The systems we consider are equipped with *Virtual Output Queues* (VOQ) at each input and have a single FIFO queue at each input. Specifically, each input i maintains a VOQ V_{ij} containing all the cells it has that are to be transferred to output j, and each output j maintains a queue Q_j of cells waiting to be sent on its outgoing link. Since a crossbar can accept at most one cell at a time from each input, and deliver at most one cell at a time to each output, there is some system-level coordination required to determine which cells are transferred when. This is done using a crossbar controller that implements a scheduling algorithm. We say that a scheduling algorithm is *work-conserving* if it does not allow any output link capacity to be wasted. More precisely, work-conservation implies that whenever there is a cell for output j in any VOQ, some cell is being sent on output j.

As its name suggests, the LOOFA scheduling algorithm matches inputs to outputs, while giving priority to those outputs with the fewest cells in their out-going queues. This is a natural greedy strategy, if our objective is to keep output links busy whenever possible. We can view the LOOFA scheduler as operating iteratively, processing the outputs in non-decreasing order of their queue lengths. When an output is considered, it is matched with an input that has not yet been matched to any output, and that has a cell to transfer to the current output. When multiple inputs are available to match a given output, a variety of selection criteria can be applied. The worst-case performance guarantee for systems with speedups of 2 or more, is independent of the criteria used to select inputs. However, the input selection criteria can become important for systems operated with smaller speedup; in these cases, criteria that seek to provide fairness among the inputs are natural choices.

We describe a family of *Generalized LOOFA* (GL) schedulers and show that any GL scheduler is work-conserving when operated with a speedup of 2. Although, the ALOOFA scheduler is not a GL scheduler, our analysis provides some interesting insights into its operation. For clarity, we view a crossbar as operating in discrete phases. During an *arrival phase* cells are received at the inputs and placed in VOQs. During a *transfer phase*, cells are transferred through the crossbar from VOQs to output queues. During a *departure phase*, cells are removed from output queues and sent on the outgoing links. Systems with speedup of 2 have two transfer phases in each operational cycle. We assume that the transfer phases occur between the arrival and departure phases.

The GL schedulers all maintain a list of active VOQs for each input. When a VOQ V_{ij} becomes active (that is, transitions from empty to non-empty), it is inserted at some position in the list for input i. When a VOQ becomes inactive, it is removed from its input's list. A GL scheduler inserts a newly active VOQ V_{ij} in the list using the following *insertion policy*.

> Insert V_{ij} either at the beginning of the list or immediately following any V_{ih} for which the number of cells in Q_h is less than or equal to the number in Q_j.

A GL scheduler selects VOQs from which to transfer cells through the crossbar by performing the following *VOQ selection step* for each input i.

> Select the first V_{ij} in the VOQ list at input i that does not conflict with any VOQ selected in an earlier step.

Here, a conflicting VOQ is simply one that contains cells for the same output. A GL scheduler may also re-order the VOQs in its lists by performing the following *compare-and-swap* operation.

> Select any two active VOQs V_{ih} and V_{ij} for which V_{ih} immediately precedes V_{ij} in the scheduling list at input i. If the number of cells in Q_j is less than the number in Q_h, exchange the positions of V_{ih} and V_{ij} in the list.

Note that there is no requirement that a GL scheduler perform any compare-and-swaps. Also note that the LOOFA scheduler can be viewed as the special case of the GL scheduler in which we precede the VOQ selection step at input i with enough compare-and-swap operations to fully sort the list according to the number of cells in the output queues.

To facilitate the analysis of the GL schedulers, we introduce some notation. First, let v_{ij} be the number of cells in V_{ij} and let q_j be the number in Q_j. Let p_{ij} be v_{ij} plus the number of cells in all VOQs that come before V_{ij} in the VOQ list at input i. Define $slack_{ij}=q_j - p_{ij}$. Note that anytime output j fails to send a cell while V_{ij} contains a cell, $slack_{ij}$ must be less than zero. Consequently, we can show that a scheduler is work-conserving by establishing that $slack_{ij}$ cannot be negative at the start of a departure phase. The following lemmas enable us to do that.

Lemma 1. If V_{ij} is active immediately before and after a transfer phase, then $slack_{ij}$ increases by at least 1 during the transfer phase.

proof. If V_{ij} or one of the VOQs that comes before V_{ij} in the input list at input i is selected during the transfer phase then p_{ij} is reduced by 1. If neither V_{ij} nor one of the VOQs that precedes it is selected, then some other input must transfer a cell to output j during the transfer phase, causing q_j to increase by 1. In either case, $slack_{ij}$ increases by 1. ■

Next, let's consider the effect of compare-and-swap operations.

Lemma 2. A compare-and-swap operation that exchanges the positions of VOQs V_{ih} and V_{ij} increases the value of $\min\{slack_{ih}, slack_{ij}\}$ by at least 1.

proof. Assume that V_{ih} comes before V_{ij} in the list at input i and note $q_h>q_j$. Also note that swapping V_{ih} and V_{ij} reduces p_{ij} and hence increases $slack_{ij}$. A swap does increase p_{ih} (hence reducing $slack_{ih}$), but the new value of p_{ih} is the same as the original value of p_{ij}, and since $q_h>q_j$, the new value of $slack_{ih}$ is larger than the old value of $slack_{ij}$. Hence the new values of both $slack_{ih}$ and $slack_{ij}$ are at least 1 larger than the original value of $slack_{ij}$, which is at least as large as the original value of $\min\{slack_{ih}, slack_{ij}\}$. ■

If we let $minSlack_i=\min_j slack_{ij}$, Lemma 2 allows us to conclude that a compare-and-swap cannot cause $minSlack_i$ to decrease and consequently, a whole series of compare-and-swaps can produce no net decrease in $minSlack_i$. Lemma 1 implies that $minSlack_i$ increases by 1 during each transfer phase. Also note that a departure phase causes $minSlack_i$ to decrease by at most 1. These observations lead to our next lemma.

Lemma 3. In a crossbar with a speedup of 2 and a GL scheduler, if any VOQ is active at input i just before a departure phase, then $minSlack_i \geq 1$.

proof. The proof is by induction on the time step t. Note that when t=0, all VOQs are empty and so the claim is trivially satisfied. Assume then that the claim is true at all time steps that precede t. This implies that if there are any active VOQs at input i at the start of time step t, then $minSlack_i \geq 0$.

If no new VOQ becomes active during the arrival phase of step t, then $minSlack_i$ can decrease by at most 1 during the arrival phase and hence, $minSlack_i \geq -1$ right after the arrival phase (assuming some VOQ is active). So assume that some V_{ij} became active during the arrival phase of time step t. If V_{ij} was inserted at the front of the VOQ list at input i then immediately after the arrival phase, we have $slack_{ij} \geq -1$ and hence, $minSlack_i \geq -1$. Now suppose that V_{ij} was inserted immediately after V_{ih}. Since $minSlack_i \geq 0$ just before the arrival phase, $slack_{ih} \geq 0$ also. Since, V_{ij} was inserted after V_{ih}, $slack_{ih} \geq 0$ after the arrival phase as well. Since $q_h \leq q_j$ (by the GL insertion policy) and v_{ij}=1, $slack_{ij} \geq slack_{ih} - 1$ following the arrival phase, implying that $minSlack_i \geq -1$.

So, in all cases, $minSlack_i \geq -1$ after the arrival phase in step t or there is no active VOQ. Since each of the two transfer phases in step t increases $minSlack_i$ by 1 and any compare-and-swaps performed by the scheduler do not decrease $minSlack_i$, we have $minSlack_i \geq 1$ before the departure phase in step t or there are no active VOQs. ■

Since, $minSlack_i \geq 1$ before any departure phase for which input i has active VOQs, $slack_{ij} \geq 1$ for all active VOQs. This yields the work-conservation result stated in the following theorem.

Theorem 1. All GL schedulers are work-conserving.

It's interesting to note that *no compare-and-swap operations are required for work-conservation*. Lemma 2 shows that they can't do any harm, but they are not necessary. The only thing that is really essential is the insertion policy. The LOOFA scheduler effectively follows this insertion policy, but takes the additional step of keeping the VOQs sorted by output occupancy. Because LOOFA maintains a fully sorted order, the VOQ lists at all inputs have a consistent ordering, although different inputs may have different sets of active VOQs. This means that a LOOFA scheduler can maintain a single list of outputs, rather than a separate list for each input.

Because LOOFA can use a single list when making scheduling decisions, it's possible to perform the required matching of inputs to outputs using a simple and fairly fast circuit and this was one of the key motivations for the ALOOFA scheduler described in [16]. However, in ALOOFA, the list is only *approximately ordered*. Specifically, the ALOOFA scheduler performs a prescribed set of compare-and-swap operations that make it "more sorted" but not fully sorted. It limits the number of compare-and-swaps done in order to enable high speed operation. Now, based on our earlier analysis, one might jump to the conclusion that the approximate sorting done by ALOOFA does not prevent us from achieving work-conservation. However, it does. ALOOFA is not a GL scheduler, because it effectively violates the GL insertion policy. When a VOQ V_{ij} becomes active in ALOOFA, its output is not repositioned in the single output list. So, if output j's immediate predecessor in the list has more cells in its output queue than output j does, we have a violation of the insertion policy. Such violations tend to get quickly "repaired" by the compare-and-swap operations done by ALOOFA, but they remain violations.

Hence, we cannot conclude that ALOOFA is work-conserving, and indeed it's easy to find traffic patterns that demonstrate that it's not work-conserving. Still, it is tantalizingly close, and its practical advantages make it a worthwhile option in real systems, which are often operated with a speedup substantially smaller than the 2× needed for worst-case performance guarantees anyway.

3. APPROXIMATE LOOFA

Here, we review the approximate LOOFA crossbar scheduler, first introduced in [16] and fill in certain details that were just sketched in the original paper. However, before describing ALOOFA, we need to describe the LOOFA scheduler on which it is based. As its name implies, the LOOFA scheduling algorithm matches inputs to outputs, while giving priority to those outputs with the fewest cells in their out-

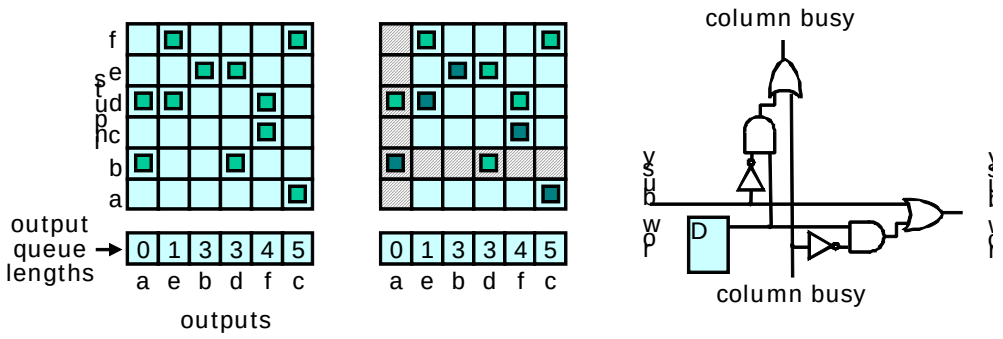

Figure 1. Approximate LOOFA matcher

going queues. This is a natural greedy strategy, if our objective is to keep output links busy whenever possible. We can view the LOOFA scheduler as operating iteratively, processing the outputs in non-decreasing order of their queue lengths. When an output is considered, it is matched with an input that has not yet been matched to any output, and that has a cell to transfer to the current output. When multiple inputs are available to match a given output, a variety of selection criteria can be applied. The worst-case performance guarantee for systems with speedups of 2 or more, is independent of the criteria used to select inputs. However, the input selection criteria can become important for systems operated with smaller speedup; in these cases, criteria that seek to provide fairness among the inputs are natural choices.

The ALOOFA scheduler was inspired by the observation that the matching required for LOOFA could be implemented efficiently in hardware by a relatively simple $N \times N$ array of circuit elements. The key idea is illustrated in Figure 1. The left-most part of the figure shows a 6×6 matcher array. Each element of the matcher array corresponds to an input-output pair, and stores one bit of information, indicating whether or not the given input has cells to send to the given output. In the figure, input-output pairs with cells waiting to pass through the crossbar are indicated by squares within the matcher array cells. So for example, input *e* has cells to send to outputs *b* and *d*.

Note that below the matcher array, the output queue lengths are shown. These are maintained in sorted order with the shortest queue lengths on the left. The output labels indicate which output is associated with a given queue length. So for example, output *b* has 3 cells, while output *f* has 4. The matcher array attempts to match inputs to outputs by processing cells from left-to-right and from bottom-to-top. A given cell can be matched so long as there is no other matched cell to its left in the same row or below it in the same column.

The center portion of the figure illustrates such a match. The match from input *b* to output *a*, near the lower left of the array, eliminates from contention any other matches in the first column from the left and the second row from the bottom. The other dark squares in the diagram indicate other matches. These matches can be found using the simple combinational circuit shown in the right portion of the figure. This circuit is repeated for every cell, with the *D* flip flop being set to a 1 if (and only if) that cell is *active*, meaning that its input has one or more cells for its output. The right-hand *row busy* signal is high if either the left-hand signal is asserted, or if the cell is active and the incoming *column busy* signal is low. Similarly, for the the outgoing *column busy* signal. Since these signals always flow up and to the right, the time required to find a matching is about $2n$ gate delays. For a modern ASIC process with gate delays of under 50 ps, a 32 port matcher circuit requires less than 2 ns to find a maximal matching.

To maintain the outputs in approximate sorted order, the ALOOFA controller performs a single odd-even sorting step as part of each operational cycle. Before the swap is performed, the queue lengths are adjusted to reflect transfers of cells into or out of an output queue (note that queue lengths can only change by 1). Following the queue length adjustment, the queue lengths for the odd-even column pairs are compared, and if out-of-order, they are swapped. Note that the columns of the matcher array are

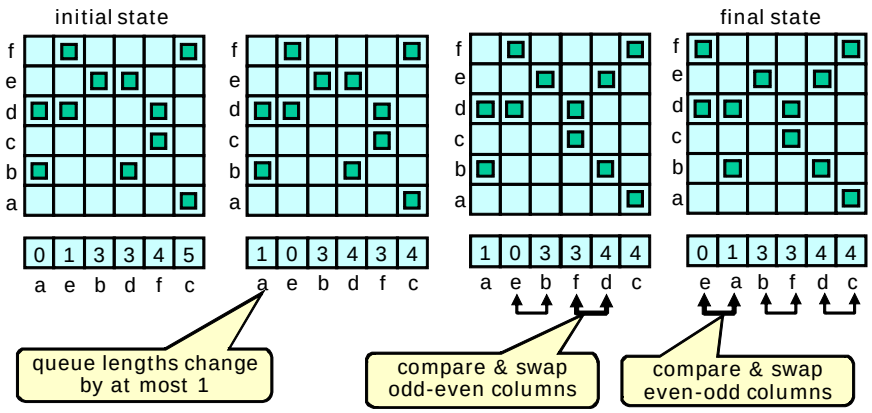

Figure 2. Odd-even swap to maintain approximate sorted order

swapped along with the queue lengths. Next, the even-odd column pairs are compared, and swapped if necessary.

In the example shown in Figure 2, the odd-even swap succeeds in restoring a sorted order. This is not guaranteed to occur in all cases, but it does occur often, since queue lengths change so little from one step to the next. Even when it does not occur, queue lengths typically do not get very far out of order. Note that it is only the sorting step that makes ALOOFA approximate. So long as outputs are sorted, the matching produced by ALOOFA is exactly the matching prescribed by LOOFA. Indeed, if the ALOOFA scheduler performed $N/2$ sorting steps, it could guarantee that the outputs remained in sorted order, but this would make it difficult to achieve a high performance implementation.

Note that as presented so far, the ALOOFA matcher gives top priority to the input at the bottom row of the array. To avoid giving some inputs a systematic advantage over other inputs, we can re-order the rows of the array so as to avoid favoring any inputs over others. Reference [16] suggests doing this by applying a random shuffling of the rows. This is illustrated in Figure 3. In this scheme, adjacent rows of the array are randomly exchanged, then all are passed through a perfect-shuffle pattern. This can be done after every operational step to ensure a high degree of input fairness. An alternative to random shuffling is to simply rotate the rows. While this is simpler, it can still result in significant unfairness; if we applied row rotation to our example matcher, input *a* would still have priority over input *b* for five out of every six operational cycles.

The shuffling of the rows and columns of the matcher array creates a new issue. Since the input and output signals to the crossbar controller appear on pins at fixed positions on the controller chip, we need some way to maintain the connection between these pins and the shifting rows and columns of the matcher array. This can be handled by adding input and output crossbars, that maintain connections between fixed IO pins and dynamically changing matcher rows and columns.

In Figure 4. the filled in crosspoints are “closed” indicating a connection between the corresponding

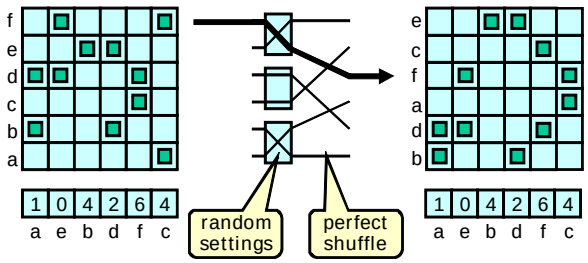

Figure 3. Row shuffling

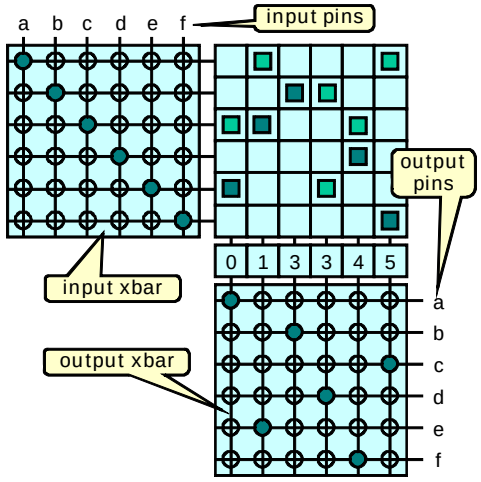

Figure 4. Matcher with IO crossbars

row and column. By swapping the rows of the input crossbar along with the rows of the matcher, we maintain the correct relationship between the IO pins and the matcher rows. Similarly, by swapping the columns of the output crossbar along with the columns of the matcher, we maintain the correct relationship between the matcher columns and the output pins.

4. DETAILED DESIGN

Reference [16] provides a good case for ALOOFA as a practical crossbar scheduler, but provides too little detail to enable a serious evaluation. A complete crossbar scheduler must interface with input and output line cards, and perform all the individual steps required by the algorithm in an appropriate sequence. The handling of these details can have significant impact on the circuit complexity and the achievable performance. In this section, we describe an actual implementation of the ALOOFA crossbar scheduler. This design has been fully specified in VHDL, and synthesized for implementation on a Xilinx Virtex 5 FPGA. We describe the complexity, performance and scaling characteristics of the synthesized circuit.

4.1 Input and Output Interfaces

The controller has an input-side interface, which provides signals used to communication with input line cards in a router or switch, and an output-side interface, providing signals for communication with output line cards. The input-side interface includes the following signals.

- **`onOff`** - This is an N bit signal used to turn on turn on and off the data present bits in the matcher array of the crossbar controller. Specifically, during the first clock tick of the controller's operational cycle, `onOff(i)` is asserted to indicate that input `i` has cells for the output specified by `target(i)`, so the matcher's data present bit indicating the presence of traffic from input `i` to output `target(i)` should be set. During the second clock tick of the controller's operational cycle, `onOff(i)` is asserted to indicate that it no longer has cells for the output specified by `target(i)`, so the matcher's data present bit indicating the presence of traffic from input `i` to output `target(i)` should be cleared. Note that for each input, there is at most one cell arrival and one cell departure per operational cycle, so most of the data present bits in the matcher remain the same from one cycle to the next.
- **`target`** - This is an N word signal, where each word specifies a crossbar output. Its use was described in the previous paragraph.
- **`grant`** - This is an N bit signal; `grant(i)` is asserted if input `i` has been selected to send a cell.
- **`sendTo`** - This is an N word signal; when `grant(i)` is asserted, `sendTo(i)` identifies the output that input `i` should send a cell to.

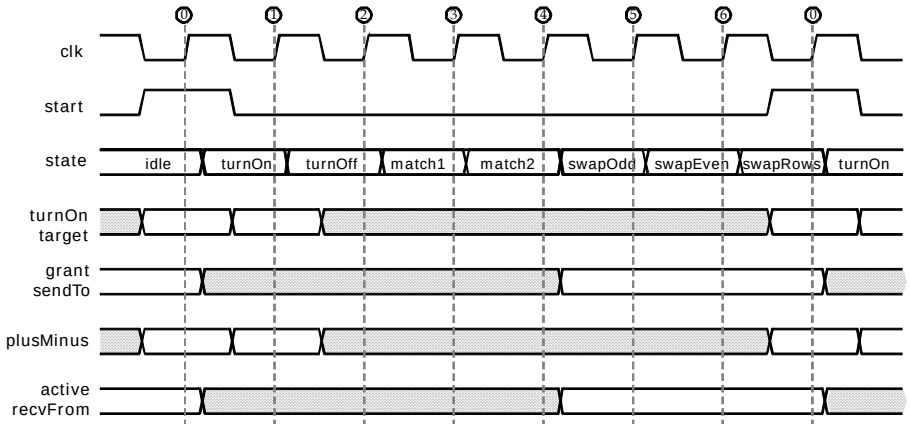

Figure 5. Timing of IO and internal operations

The output interface includes the following signals.

- **`plusMinus`** – This is an N bit signal; `plusMinus(j)` is asserted to indicate that the queue length for output `j` should be increased or decreased by 1.
- **`active`** – This is an N bit signal; `active(j)` is asserted if output `j` has been selected to receive a cell.
- **`recvFrom`** – This is an N word signal; if `active(j)` is asserted, `recvFrom(j)` identifies the input that will send a cell to output `j`.

We note that other interface definitions are certainly possible, and that the choice of interface can have a significant effect on both circuit complexity and performance. We discuss some of these alternatives below.

4.2 Operational Cycle

The ALOOFA scheduler operates on a periodic schedule extending over seven clock periods, as shown in Figure 5. The first two clock periods are used for receiving data from the input and output line cards. Specifically, on the rising edge of clock tick 0, the input line cards signal which matcher cells should "turn on" (if any) and during clock tick 1, they signal which matcher cells should turn off. Concurrently, the output line cards signal which queue lengths should be increased by one, and which should be decreased by one.

The next two clock ticks are used to match inputs to outputs. Two clock ticks are allocated to the matching to allow time for signals to propagate through the complete matching array. There are two approaches one can take to implementing this. One approach is to allow the signals to simply propagate through the matcher over two cycles and use the output signals from matcher at the end of the second clock cycle. Implementing this approach in the context of modern CAD tools suites, requires the explicit identification of so-called *multi-cycle circuit paths* to inform circuit synthesis tools that these paths can tolerate larger than normal circuit delays. It turns out that this is a relatively tedious manual process, in most CAD tool suites. Consequently, we have adopted the common practice of introducing a bank of pipeline registers to explicitly break the combinational circuit paths into two parts that are handled during different clock ticks. This allows the CAD tools to treat all circuit paths in a uniform way. In the case of the matcher, the pipeline registers are positioned along the top-left to bottom-right diagonal of the matching array.

Following the matching process, an odd-even sorting step is performed over two clock ticks. After this, rows are swapped randomly, using the random exchange and shuffle procedure described earlier.

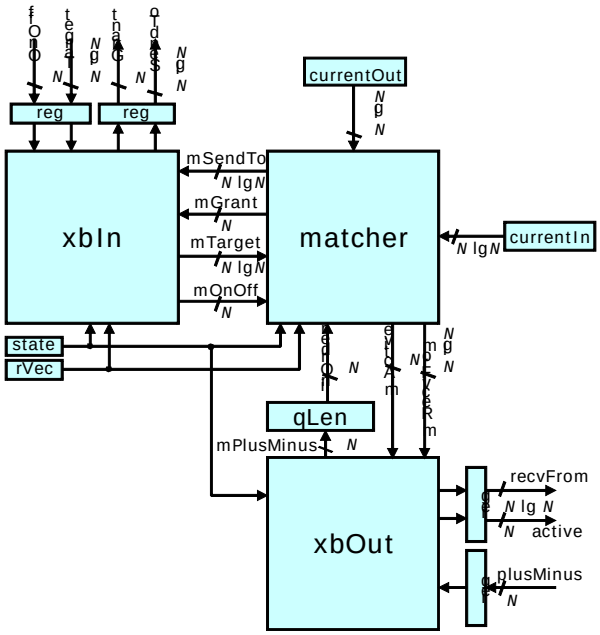

Figure 6. Detailed block diagram

4.3 Implementation Details

The block diagram in Figure 6 provides a more detailed picture of the major components of the crossbar controller and some of the key internal signals linking various components. The input-side interface is shown at the top left. Note that the arriving inputs and departing outputs all pass through clocked registers at the interface. The same is true for the output-side interface signals at the bottom right. Also note that the signals on the input and output side have corresponding signals at the interface to the matcher, that are propagated through the input and output crossbars. The matcher's version of these signals are all prefixed with an 'm' to distinguish them from the external interface signals.

The *state* register controls the timing of the operations performed by the various components, as illustrated in Figure 5. The *rVec* register implements a 32 bit linear feedback shift register and the lower *N*/2 bits of it are used to control row swapping in the input crossbar and matcher. The *qLen* register, contains the output queue lengths, arranged in sorted order. It has *N* inputs that are used to either increment or decrement the stored values. It also has *N* outputs, each of which controls whether a column swap is required between one column and the next. The `currentIn` register specifies which input is currently associated with a given row of the matcher array. Similarly, the `currentOut` register specifies which output is currently associated with a given column of the matcher array. These signals are maintained by the top-level controller and are updated whenever rows or columns are swapped.

Given this context, it's instructive to see how the main elements of the matcher are specified using VHDL. Let's start with the combinational logic used to define the matches between inputs and outputs.

```
for i in 0 to N-1 loop
   colBusy(i) <= (others => '0'); rowBusy(i) <= (others => '0');
end loop;
rowBusy(N) <= (others => '0');
for i in 0 to N-1 loop
   for j in 0 to N-1 loop
      if i+j /= N-1 then
         match := dp(i)(j) and (not rowBusy(i)(j)) and (not colBusy(i)(j));
         rowBusy(i+1)(j) <= rowBusy(i)(j) or match;
         colBusy(i)(j+1) <= colBusy(i)(j) or match;
      else
```

```
            match := dp(i)(j) and (not rbDiag(i)) and (not cbDiag(i));
            rowBusy(i+1)(j) <= rbDiag(i) or match;
            colBusy(i)(j+1) <= cbDiag(i) or match;
         end if;
      end loop;
   end loop;
```

The `colBusy` and `rowBusy` signals are produced and used by successive cells in the matcher array. Specifically, `colBusy(i)(j)` is an input to the (*i*,*j*) cell in the array and `colBusy(i)(j+1)` is an output. Similarly, `rowBusy(i)(j)` is an input to the (*i*,*j*) cell in the array and `rowBusy(i+1)(j)` is an output. The match logic is defined in the inner loop. There are two cases, with the first case applying to most of the cells. In this case, the match variable is high if the data present bit for the (*i*,*j*) cell is set and there are no matches below or to the left of the current cell (as indicated by the `colBusy` and `rowBusy` inputs). The second case applies just to the diagonal cells of the matcher array (top-left to bottom-right). These cells contain flip flops for the row and column busy signals, to limit the number of stages of logic through which signals must propagate in a single clock tick. These flip flops are defined by the `rbDiag` and `cbDiag` signals.

For those who are less familiar with VHDL, a brief word about the two kinds of assignment statements used above. *Signal assignments* use the left arrow symbol (`<=`) and such assignments define actual circuit connections. *Variable assignments* use the traditional Algol assignment symbol (`:=`) and are best thought of as macro definitions, that allow logic specifications to be stated more concisely. Also, note that the expression `(others => '0')` is a common idiom in VHDL, that is used to specify that all bits of a multi-bit signal of unspecified length are '`0`'.

Next, let's move onto the logic that generates the `active` and `recvFrom` outputs. The signal `mActive(i)` is obtained directly from the `colBusy` signals generated earlier. The generation of the signal `mRecvFrom` is a bit more complicated. For each column in the array, its value should be the input corresponding to the matching cell in that column (assuming there is one). In the following code fragment, the inner loop specifies `xRecvFrom` as the logical-OR of the values of `currentIn(j)` for all cells in column `i` that correspond to a match. Since there is at most one cell per column that can have a match, `xRecvFrom` is the input corresponding to that matching cell.

```
for i in 0 to N-1 loop
   xRecvFrom := (others => '0');
   for j in 0 to N-1 loop
      match := dp(i)(j) and (not rowBusy(i)(j)) and (not colBusy(i)(j));
      if match = '1' then
         xRecvFrom := xRecvFrom or currentIn(j);
      end if;
   end loop;
   mActive(i) <= colBusy(i)(N); mRecvFrom(i) <= xRecvFrom;
end loop;
```

The logic for generating the `grant` and `sendTo` outputs is similar.

Now, let's consider the process that responds to operations specified by the top-level controller. The main part of this process is a large case statement, with a separate case for each state. The case for the `turnOn` state is shown below.

```
when turnOn =>
   for i in 0 to N-1 loop
      for j in 0 to N-1 loop
         if mOnOff(j) = '1' and currentOut(i) = mTarget(j) then
            dp(i)(j) <= '1';
         end if;
      end loop;
   end loop;
```

The case for the `turnOff` state is similar. The matcher loads the pipeline registers along the diagonal of the matcher array at the end of the `match1` state.

```
when match1 =>
   for i in 0 to N-1 loop
      rbDiag(i) <= rowBusy(i)((N-1)-i);
       cbDiag(i) <= colBusy(i)((N-1)-i);
   end loop;
```

During the `swapEven` and `swapOdd` states, the matcher swaps adjacent columns if they are not in the correct order.

```
when swapEven =>
   for i in 0 to (N/2)-1 loop
      if inOrder(2*i) = '0' then
         dp(2*i) <= dp(2*i+1); dp(2*i+1) <= dp(2*i);
      end if;
   end loop;
when swapOdd =>
   for i in 1 to (N/2)-1 loop
      if inOrder(2*i-1) = '0' then
         dp(2*i) <= dp(2*i-1); dp(2*i-1) <= dp(2*i);
      end if;
   end loop;
```

Note that because VHDL specifies circuit connections, not sequential execution, the column swapping specified by the pair of assignments is a single operation that takes place at one time, so there is no need for a temporary register in which to store the value, as would be required in a sequential program.

Finally, when in the `swapRows` state, the matcher does a random exchange of adjacent rows, based on the bits of the `rVec` input signal. It then shuffles all the rows. The variable `dpSwap` represents the intermediate values in this two step process.

```
when swapRows =>
   for i in 0 to N-1 loop
      for j in 0 to (N/2)-1 loop
         if rVec(j) = '0' then
            dpSwap(2*j):= dp(i)(2*j); dpSwap(2*j+1):= dp(i)(2*j+1);
         else
            dpSwap(2*j) := dp(i)(2*j+1); dpSwap(2*j+1) := dp(i)(2*j);
         end if;
      end loop;
      for j in 0 to N-1 loop
         if j < N/2 then
            dp(i)(2*j) <= dpSwap(j);
         else
            dp(i)(2*(j-N/2)+1) <= dpSwap(j);
         end if;
      end loop;
   end loop;
```

Note that the bulk of the matcher specification is concerned with overhead activities, rather than the essential matching operation. This is also true of the circuitry generated by the specification.

4.4 Functional Simulation

To demonstrate the operation of the circuit we present results of a functional simulation for an eight port configuration of the ALOOFA scheduler. During the first operational cycle of the controller, the simulation inputs specify the turning on of data present bits for all inputs to output 0. During the second operational cycle, the simulation inputs specify turning on data present bits for inputs 1..7 to output 1. During the third operational cycle, the simulation inputs specify turning on data present bits for inputs 2..7 to output 2, and the pattern continues in this fashion.

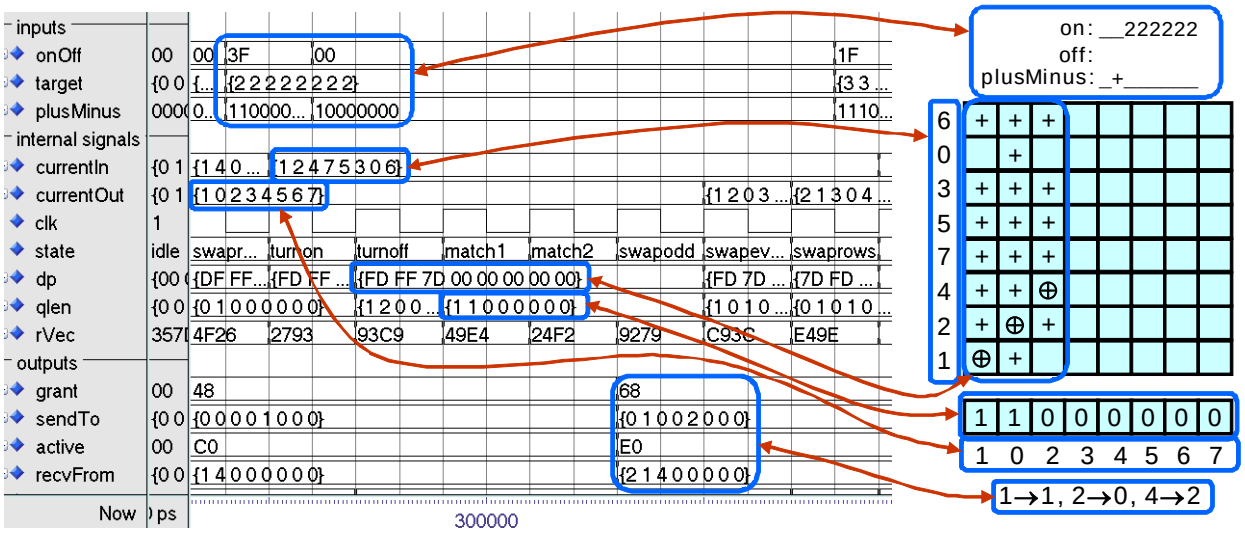

Figure 7. Sample simulation output

Figure 7 shows a snapshot of the simulation during the third operational cycle. The diagram at the right displays information obtained from the simulation output in a graphical form to facilitate understanding. The main array of squares represents the matcher array. The arrows linking parts of the simulation output to the diagram show how the two correspond. Starting at the top, we see that the `onOff` signal is 3F during the first tick of the controller's operational cycle and all fields of the `target` signal are 2; this specifies the turning on of data present bits for inputs 2..7 to output 2. We also note that the output queue lengths are incremented for outputs 0 and 1, but also decremented for output 0. The `currentIn` signal determines the effective ordering of the rows of the matcher array, while the `currentOut` signal determines the column ordering. The state of the matcher array is determined by the `dp` (data present) bits. Each pair of hex digits in the `dp` bits represents one column of the array, so the first pair of digits (`FD`) specifies the first column of the array and indicates that all bits in this column are set, except for the second one from the top (left-to-right ordering in the `dp` bits corresponds to bottom-to-top ordering in the graphic). Simillarly, the subsequent pairs of hex digits (`FF` and `7D`) specify the next two columns. In the graphic, active cells are indicated by a + symbol and those that are selected during the matching process are indicated by a ⊕ symbol. The results of the matching of inputs to outputs is reflected in the `sendTo` and `recvFrom` signals, along with the `grant` and `active` signals. These are shown in the graphic as a list of matched pairs.

4.5 Circuit Complexity

We evaluated the circuit complexity of the ALOOFA controller by synthesizing the circuit for a Xilinx Virtex 5 FPGA, with 6 input LUTs (specifically, the XC5VFX30-2 part). The circuit complexity is shown in the table below, with separate columns shown for the number of Lookup Tables (LUTs) and flip flops (FFs). We give data for 8×8 and 16×16 circuit configurations and use these to compute an empirical scaling factor.

	VHDL lines	8x8		16x16		scaling factor	
		LUTs	FFs	LUTs	FFs	LUTs	FFs
matcher	160	745	79	2,975	288	4.0	3.6
input crossbar	95	307	64	1,507	256	4.9	4.0
output crossbar	79	240	64	1,053	256	4.4	4.0
top	232	941	363	1,752	786	1.9	2.1
total	614	2,233	570	7,287	1,586	3.3	3.6

We separate results for the matcher, the crossbars and the top level circuit, to clarify the relative contributions of different parts of the circuit. We include empirical scaling factors at the right and note that (as expected), the matcher and crossbar components exhibit roughly quadratic scaling, while the top level circuit exhibits roughly linear scaling. In smaller configurations, the top level circuit dominates the hardware complexity, but contributes a smaller fraction in larger configurations. The matcher is substantially larger than the input and output crossbars. We note that the core matching operation performed by the matcher can be implemented with just two LUTs per cell, as can be seen from Figure 1. The actual number of LUTs is about five times larger than required by the matching operation alone. The remaining LUTs are used to generate the output signals (`sendTo`, `recvFrom`) and to perform the row and column swapping operations. We also note that that for these (relatively modest) values of N, the overall circuit complexity grows more slowly than one would expect, based on the asymptotic circuit complexity of $O(N^2 \log N)$. This is primarily due to to the large role played by the top level controller. Since its complexity is essentially linear in N, it has a relatively large impact when N is small.

It's also worth noting that the FPGA used for these results has 19,200 LUTs and 19,200 flip flops that can be used to implement circuitry, so even the 16×16 crossbar uses a modest fraction of the available resources. Moreover, this is a relatively small FPGA. Finally, we note that the circuit has a high ratio of LUTs to flip flops (more than 4:1 overall, and roughly 10:1 in the matcher). This is mainly a consequence of the fact that the matcher and crossbars, require relatively few flip flops, relative to combinational circuitry. Finally, we note that while the specification of the controller is reasonably concise, requiring a total of 614 lines of VHDL (about 12 pages); the top level circuit accounts for just over a third of the total, while the matcher accounts for just over one fourth.

4.6 Circuit Performance

To evaluate the performance of the circuit, we performed a complete place-and-route of the 8x8 version of the ALOOFA controller, with a timing constraint on the clock period. The smallest clock period for which the automated tools were able to successfully complete the place and route process was 5.7 ns. Since, a complete operational cycle of the controller takes 7 clock ticks, this translates to an overall cycle time of 40 ns. This is fast enough for a 50 byte packet at 10 Gb/s. For 10 Gb/s Ethernet, the smallest effective frame size is about 80 bytes (including the standard preamble and required inter-frame spacing), so the FPGA implementation of the ALOOFA controller can support a 10 GbE switch with a speedup of

1.6, while processing minimum size packets. It can provide a 2:1 speedup for frame lengths of at least 100 bytes.

Larger versions of the circuit cannot match that performance using FPGA technology. In general, one would expect that each doubling of the crossbar size would result in a doubling of the minimum clock period, and hence halving of the operating frequency. So for example, we would expect an FPGA implementation of a 32x32 controller to have a clock period of about 22.8 ns and a resulting operating frequency of about 43 MHz. An ASIC implementation can be expected to bring the clock period down by an order of magnitude, enabling an operating frequency of more than 400 MHz. This would support a 2:1 speedup with minimum size packets with a comfortable margin.

There are several ways one can improve performance, further. The results provided above are based on a fully automated place-and-route, with no explicit floor-planning to guide the tools to the most efficient layout. Given the highly regular nature of the circuit, it's likely that significantly better results could be obtained through careful floor-planning.

There are also higher level changes one might make to improve the design. In particular, for larger circuit configurations, the introduction of additional pipeline registers could have a big impact on the operating frequency. The introduction of pipeline registers at the interfaces to the matcher can be expected to improve the clock frequency in a larger configuration by a factor between 1.5 and 2. This would add an additional clock tick to the operational cycle, reducing the gain by about 15%. Still, the improvement would be worthwhile.

Another way to improve the performance is to combine the first two steps of each operational cycle. This would increase the number of pins required by $N(2+\log_2 N)$, but would reduce the number of ticks per cycle from 7 to 6. One could also reduce the number of clock ticks used for row and column swapping. For example, if row swapping were not done on every operational cycle, we could improve the operational performance, at the cost of a reduction in short-term fairness. One could also do only one column swap each cycle, alternating between the odd and even swap steps, although this could have an impact on the scheduling performance. Combining all such optimizations, one might be able to reduce the number of clock ticks per cycle from 7 to 4, yielding an improvement of 75% in operating frequency.

5. SCHEDULING PERFORMANCE

Since the late nineties, it has been known that certain crossbar schedulers could provide strong performance guarantees when used with a speedup of 2 or more [3,6]. While it has generally been assumed that a non-trivial speedup was necessary to achieve such guarantees, this has not been proven in any definitive way. In this section, we introduce a natural lower bound on the performance of any crossbar scheduler and use it to show that certain traffic patterns require a non-trivial speedup, in order to achieve work-conserving performance. We then compare the performance of LOOFA and ALOOFA to this lower bound and demonstrate that both closely match the lower bound and that their performance is nearly identical.

We define a traffic pattern to be a schedule of cell arrivals at the inputs of a crossbar. Each arriving cell has a designated output, and the job of the crossbar scheduler is to decide when arriving cells should be transferred from inputs to outputs. We can define two variants of the crossbar scheduling problem. In the *offline scheduling* problem, the entire traffic pattern is known in advance and the scheduler can use its knowledge of the complete traffic pattern in making scheduling decisions. In the *online scheduling* problem, the scheduler must make scheduling decisions as cells arrive, with no advance knowledge of the complete traffic pattern. In practice, we are most interested in online scheduling, but the offline problem is useful in establishing broadly applicable lower bounds.

We assume a synchronous switch model in which cells arrive and depart at integer times $i \geq 0$ and that for a switch with a speedup of S, cells are transferred at times i/S for integers $i \geq 0$; we also assume that a cell that arrives at time t, can be transferred through the switch at time t if t is an integer multiple of $1/S$.

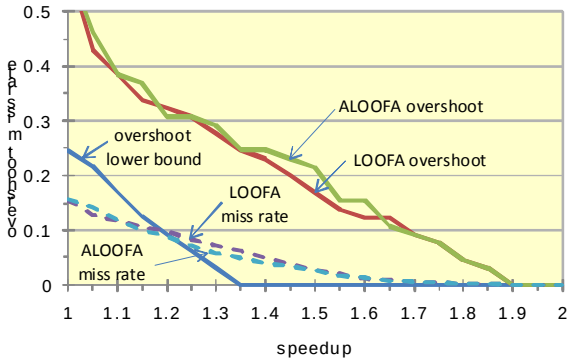

Figure 9. Scheduler Performance

However, it cannot be forwarded at the same time it is transferred though the crossbar. This implies that every cell is delayed by at least one time step. For any traffic pattern, P, we let P_t be the subset that includes cells that arrive at times $\leq t$. Any such subset can be viewed as defining a bipartite graph on the crossbar's inputs and outputs. We note that the number of cells that can be transferred through a crossbar at time t is bounded above by the size of a maximum size matching on the graph defined by P_t. This observation is the basis for our lower bound. If n is the number of cells in a traffic pattern P and we define m_t to be the size of a maximum size matching on the graph defined by P_t, then a schedule that transfers all cells through the crossbar by time t must satisfy the following condition.

$$\sum_{0 \leq i \leq tS} m_{i/S} \geq n$$

since the schedule cannot finish until all cells are transferred. The smallest value of t that satisfies this criterion is thus a lower bound on the time to transfer all cells through the crossbar, and the next larger integer is a lower bound on the time to both transfer and forward all cells in the schedule.

We can apply our lower bound to a traffic pattern proposed in [16] as an effective *stress test* for crossbar schedulers. This provides concrete evidence that this traffic pattern really is inherently challenging. The traffic pattern is defined on n inputs and $2n-1$ outputs. Let $0 \leq i < n$ and $0 \leq t < n$ be integers. At time t, all inputs $i \geq t$ receive a cell for output t, while all inputs $i < t$ receive a cell for output $n+i$. This is illustrated in Figure 8 for the case of $n=4$ (inputs and outputs are assumed to be numbered from 0, starting at the top). Note that in this case, a crossbar with a speedup of 1 can transfer only one cell at time 0 and three cells at time 1. After that, it can transfer at most four cells at a time. Since a total of 16 cells are received, it's not possible to transfer and forward all cells until time 5, while an ideal output-queued switch could transfer and forward all cells by time 4. Note that this is true no matter what scheduling algorithm is used. We define the excess time used by a scheduler, relative to the completion time of an ideal output-queued switch as the *overshoot* of the schedule. So, for this traffic pattern, the best possible overshoot is 25%.

Lower bounds can be computed for the stress test traffic pattern for any speedup. Figure 9 shows how the lower bound varies with speedup for stress test traffic patterns with $n=64$. The overshoot drops as the speedup increases, and is zero for speedups larger than about 1.33. We observe that the completion times for the LOOFA and ALOOFA schedulers are generally within 25% of the completion time lower bound. There is a significant gap between the lower bound and the performance of LOOFA and ALOOFA; this gap defines the potential space for improvement of either the lower bound or the schedulers. We observe that there is very little difference between the performance of LOOFA and ALOOFA, and that in fact, ALOOFA performs slightly better than LOOFA in some cases. This confirms the intuition that the approximate sorting does not have a significant impact on the performance.

Figure 9 shows the performance of LOOFA and ALOOFA using another performance metric, in addition to the overshoot. The *miss rate* for a schedule is defined to be the fraction of times that an output is

unable to send a cell, even when there is a cell in the system for that output. This provides a measure of the fraction of a system's output capacity that is effectively lost due to less than ideal scheduling performance. We observe that LOOFA and ALOOFA have nearly identical miss rates and that for speedups larger than about 1.4, the amount of lost output capacity is less than 5%.

We note that while the stress test pattern does require a non-trivial speedup, it may well be the case that other traffic patterns have even larger intrinsic overshoots. We have not yet found "more stressful" traffic patterns, but note that lower bounds, such as the one described here, provide a useful tool for evaluating candidates for the title of most challenging traffic pattern.

6. CLOSING REMARKS

There are several useful directions for extending the work described here. One is to perform a more comprehensive study of the hardware performance of the ALOOFA scheduler, with a focus on ASIC synthesis of larger configurations. In this context, it would be worthwhile exploring alternate design choices, such as the impact of additional pipeline stages and/or a reduction in row/column swapping overhead. There may also be alternate strategies for approximate sorting that would perform better than the simple odd-even strategy used here.

It would also be interesting to further explore the worst-case scheduling performance of ALOOFA. We note that an arbitrary list of N values can be sorted in $N/2$ pairs of odd-even sorting steps. This suggests the while the output ordering provided by ALOOFA may not always match the exact output ordering, there may just be a finite "lag" that would still allow a slightly weaker form of worst-case performance guarantee. Specifically, one might conjecture that ALOOFA scheduler never fails to forward a cell for an output, so long as there is no cell in the system for that output that has been present for more than $N/2$ time units. An approximate work-conservation result of this sort would provide a stronger case for the application of ALOOFA in real systems.

It's also interesting to note that the structure of the ALOOFA scheduler can be adapted for use with any other crossbar scheduler that matches inputs to outputs based on a single, consistent ordering of the outputs. The ordering may depend on other parameters, in addition to the output queue length, or could be based on completely different criteria. This is because the matching array allows fast and efficient computation of maximal matchings, and requires only that the matching proceed from the bottom left to the top right of the array.

REFERENCES

[1] Anderson, T., S. Owicki., J. Saxe and C. Thacker. "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems*, 11/93.

[2] Attiya, H., D. Hay and I. Keslassy. "Packet-Mode Emulation of Output-Queued Switches," *Proc. of ACM SPAA*, 2006.

[3] Chuang, S.-T. A. Goel, N. McKeown, B. Prabhakar "Matching output queueing with a combined input output queued switch," *IEEE Journal on Selected Areas in Communications*, 12/99.

[4] Chuang, Shang-Tse, Sundar Iyer, Nick McKeown. "Practical Algorithms for Performance Guarantees in Buffered Crossbars," *Proceedings of IEEE INFOCOM*, 3/05.

[5] Iyer, S., R. Zhang, and N. McKeown, "Routers with a Single Stage of Buffering", *ACM SIGCOMM '02*, Pittsburgh, USA, Sep. 2002.

[6] Krishna, P., N. Patel, A. Charny and R. Simcoe. "On the speedup required for work-conserving crossbar switches," *IEEE J. Selected Areas of Communications*, 6/99.

[7] Katevenis, M., G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos. "Variable Packet Size Buffered Crossbar (CICQ) Switches," *Proceedings IEEE International Conference on Communications*, pp. 1090-1096, 6/2004.

[8] Katevenis, M., G. Passas. "Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures," *Proceedings IEEE International Conference on Communications*, 5/2005.

[9] Leonardi, E., M. Mellia, F. Neri, and M.A. Marsan, "On the stability of input-queued switches with speed-up," *IEEE/ACM Transactions on Networking*, Vol. 9, No. 1, pp. 104–118, February 2001.

[10] Magill, B., C. Rohrs, R. Stevenson, "Output-Queued Switch Emulation by Fabrics With Limited Memory," *IEEE Journal on Selected Areas in Communications*, pp. 606–615, 5/2003.

[11] Marsan, M. A., A. Bianco, P. Giaccone, E. Leonardi and F. Neri. "Packet-Mode Scheduling in Input-Queued Cell-Based Switches," *ACM/IEEE Transactions on Networking*, 2002.

[12] McKeown, Nick. "iSLIP: a scheduling algorithm for input-queued switches," *IEEE Transactions on Networking*, 4/99.

[13] McKeown, N., A. Mekkittikul, V. Anantharam, and J. Walrand. "Achieving 100% Throughput in an Input-Queued Switch," *IEEE Transactions on Communications*, Vol. 47, No. 8, Aug. 1999.

[14] Mhamdi, L., Mounir Hamdi. "MCBF: A High-Performance Scheduling Algorithm for Buffered Crossbar Switches," *IEEE Communications Letters*, 2003.

[15] Nojima, S., E. Tsutsui, H. Fukuda, M. Hashimoto. "Integrated Services Packet Network Using Bus Matrix Switch", *IEEE Journal on Selected Areas of Communications*, 10/87.

[16] Pappu, Prashanth and Jonathan Turner. "Stress-Resistant Scheduling Algorithms for CIOQ Switches," *Proceedings of ICNP*, November 2003.

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

[18] Rojas-Cessa, E. Oki, Z. Jing, and H. J. Chao, "CIXB-1: Combined Input-One-cell-Crosspoint Buffered Switch," *IEEE Workshop on High Performance Switching and Routing*, Dallas, TX, July 2001.

[19] Stevens, D. and H. Zhang. ``Implementing Distributed Packet Fair Queueing in a Scalable Switch Architecture," *Proceedings of Infocom*, 1998.

[20] Turner, J. "Strong Performance Guarantees for Asynchronous Buffered Crossbar Schedulers," *ACM/IEEE Transacitions on Networking*, 8/2009.