Low-Latency Scheduling in Large Switches

Wladek Olesinski, Nils Gura, Hans Eberle
Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025, USA
{wladek.olesinski, nils.gura, hans.eberle}@sun.com

Andres Mejia
Dept. de Informatica de Sistemas y Computadores
Universidad Politecnica de Valencia
P.O.B. 22012, 46022 - Valencia, Spain
andres@gap.upv.es

ABSTRACT

Scheduling in large switches is challenging. Arbiters must operate at high rates to keep up with the high switching rates demanded by multi-gigabit-per-second link rates and short cells. Low-latency requirements of some applications also challenge the design of schedulers. In this paper, we propose the *Parallel Wrapped Wave Front Arbiter* with *Fast Scheduler* (*PWWFA-FS*). We analyze its performance, present simulation results, discuss its implementation, and show how this scheme can provide low latency under light load while scaling to large switches with multi-terabit-per-second throughput and hundreds of ports.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking – *routers*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Switching, scheduling, arbiter, low latency.

1. INTRODUCTION

Large switches with hundreds of ports and terabit-per-second throughput require schedulers/arbiters (these terms are used interchangeably) that match the input ports with output ports in order to forward data at high rates. Assuming a large switch fabric for fixed-size packets (cells), one of the main challenges in designing a scheduler is providing a maximal matching between input and output ports for every slot, where a *slot* is defined as the transmission time of a cell. As link rates grow and cell sizes remain constant, the cell time (slot) decreases. As a result, the scheduler has less time to provide a maximal matching for cells arriving at the input ports. Calculating a schedule for a switch with a large number of ports is further complicated as the computation time grows with the number of ports.

When executing software applications that require tight coupling between compute nodes, low latency is important. Networks are usually engineered such that the switches operate under light load most of the time to avoid forwarding delays caused by high load. Therefore, it is important to design a switch that provides low latency specifically under low load.

In [9] we proposed the *Parallel Wrapped Wave Front Arbiter* (*PWWFA*). In this scheme, we effectively pipelined the *Wrapped Wave Front Arbiter* (*WWFA*) [10], increasing its throughput and making it practical for very large switches with hundreds of ports and multi-terabit-per-second port rates.

Other work on this subject includes iterative schemes like *PIM* [1], *iSLIP* [6], or *DRRM* [2] that find the maximal matching by iterative, input/output round-robin arbitration. They require multiple exchanges of requests and grants as well as multiple iterations of pruning requests and grants, which is a fairly time-consuming operation and therefore not applicable for switches that have more than a few dozen of 10Gbps ports.

The current approaches that attempt to improve scheduling throughput involve pipelining of iterative schemes ([7],[8]). A scheduler consists of a number of subschedulers that process several sets of cells concurrently such that in every slot, one of the subschedulers produces a match.

Other approaches like [5] involve frame-based switching. In these schemes, multiple cells are aggregated into frames and then collectively scheduled at the same time. Effectively, a switch configuration is calculated once every several packet times (i.e. frame) and remains the same until the next frame is created.

Our *Parallel Wrapped Wave Front Arbiter* (*PWWFA*) [9] has a relatively high latency under light load. In this paper, we extend it by a *Fast Scheduler* (*FS*), and propose *PWWFA-FS*. The original *PWWFA* consists of a matrix of elements that maintain and process requests for outputs. With several "waves" of processing performed concurrently, the arbiter can schedule multiple cells simultaneously. The *Fast Scheduler* proposed in this paper augments schedules produced by *PWWFA* with requests that have arrived most recently. Our simulations of a 256-port switch show that *PWWFA-FS* reduces the scheduling delay by an order of magnitude under low load. Notably, we strived to keep *PWWFA-FS practical*, and made sure that it can be implemented in current chip technology.

The paper is organized as follows. In the next section, we briefly describe our *PWWFA* scheme the new *PWWFA-FS* is based on. In Section 3, we talk about *PWWFA-FS* in detail. Section 4 highlights two methods that the *Fast Scheduler* can use, and outline their complexity. We briefly discuss the complexity of *PWWFA-FS* in Section 5, and look at its performance in Section 6. Finally, we conclude the paper in Section 7.

2. *PWWFA*: PARALLEL WRAPPED WAVE FRONT ARBITER

We assume a switch that has N input and N output ports, and that every input port (or data source) s has N *Virtual Output Queues* (*VOQ*s). Each *VOQ* stores cells destined for one of the N output ports (or destinations) d. In every slot, the arbiter has to select up to N cells, one from each input port, and match them with the output ports. In one slot, an input port can send at most one cell, and an output port can receive at most one cell.

An example structure of *PWWFA* for an $N=4$ switch is shown in Figure 1 (from [9]). Each element of this structure represents one *VOQ*. Rows correspond to input ports, and columns correspond to *VOQ*s (i.e. output ports). For example, element (2,1) represents *VOQ* 1 at input port 2. Each element contains a request counter R_{sd} that represents the occupancy of VOQ_{sd}.

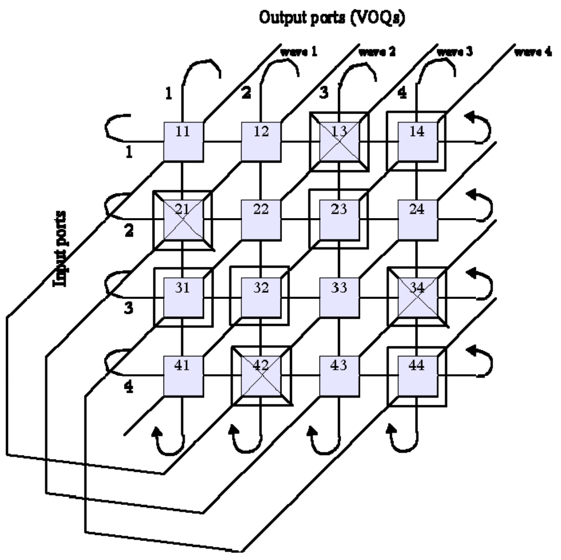


Figure 1. *PWWFA* for a 4x4 switch

We can distinguish N "waves" in this structure. Each wave comprises a set of N conflict-free requests. For example, wave 1 in Figure 1 includes elements (1,1), (4,2), (3,3), and (2,4). Since there is no conflict between them, the arbiter can process all requests in these elements at the same time. After processing of N waves starting from wave 1 and proceeding to wave N, a schedule for one slot is calculated.

When a cell arrives at VOQ_{sd}, counter R_{sd} is incremented; when a request is granted, R_{sd} is decremented. Changes of R_{sd} can occur at any time during a scheduling cycle.

The scheme employs up to N independent *subschedulers* that concurrently process different waves of the *PWWFA* matrix. Table 1 shows an example of how two subschedulers A and B process different waves of the matrix shown in Figure 1. Initially, subscheduler A starts with processing wave 1, and then proceeds to wave 2, while subscheduler B pauses. For the next two waves, A and B process elements separated by two waves. Subscheduler A finishes the scheduling cycle on the last wave (underlined 4) and produces a schedule. It then starts a new scheduling cycle also from wave 4 – beginning the scheduling cycles at different waves helps maintain fairness. Finally, subscheduler B processes wave 4, finishes its own scheduling cycle and produces a schedule.

Table 1. Parallel processing in *PWWFA*

		Waves								
Subscheduler	**A**	1	2	3	$\underline{4}$	4	1	2	$\underline{3}$	3
	B	x	x	1	2	3	$\underline{4}$	4	1	2

Thus, every two wave times, one of the subschedulers finishes processing the entire matrix and creates a schedule. This way, schedules are created twice more frequently than if only one subscheduler was used (like in the original *WWFA*). This relaxes the timing constraints imposed on the scheduler by high port rates and small cells.

The arbiter grants a request at element *(s,d)* if the following holds: (1) $R_{sd}>0$, that is, input port s requests output port d; and (2) no grant was issued in row s or column d in any of the preceding waves processed by *the same* subscheduler (see below for more on that).

When a subscheduler grants a request, it decrements the corresponding R_{sd} counter. The same counter may later be used by other subschedulers that process their own sequences of waves.

The "preceding wave" in condition (2) is understood here as any wave processed by a given subscheduler before the current wave since a scheduling cycle was started. For example, in Figure 1, wave 1 precedes waves 2, 3, and 4 that, altogether, have to be processed in order to calculate the schedule. The second condition then means that, for example, if the request in element (1,1) received a grant in the 1st wave, then requests in elements belonging to row 1 or column 1 cannot be granted by the subsequent three waves by the same subscheduler.

An example of a schedule for just one subscheduler is shown in Figure 1. Double squares represent requests, and crossed double squares represent requests that were granted.

In [9] we determine the smallest number of subschedulers needed to meet the required performance at the lowest scheduling latency for a switch with the given number of ports and link rates, and present other details of *PWWFA*.

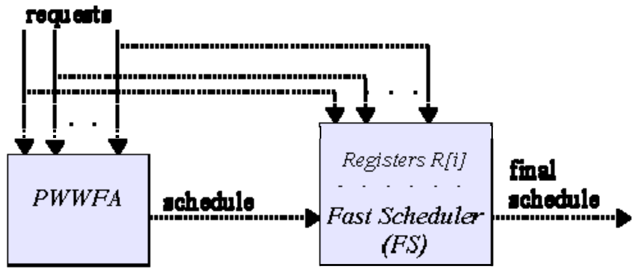


Figure 2. High-level view of PWWFA-FS arbiter

3. *PWWFA-FS*: PARALLEL WRAPPED WAVE FRONT ARBITER WITH FAST SCHEDULER

The high-level view of our new scheduler is shown in Figure 2. It consists of the *PWWFA* scheduler described above, and a *Fast Scheduler* (*FS*) module that augments the schedules created by *PWWFA*. This module uses a set of registers that keep the most recent requests.

As described in Section 2, every request that arrives at the scheduler increments the request counter R_{sd} in the appropriate *PWWFA* matrix element. In addition, this request, represented by identifier d of the requested output port, is stored in one of *N* registers *R[s]*, $0<s\leq N$, where *N* is the number of ports and *s* is the index for an input. Each *R[s]* stores only one request, so every new request from an input *s* overwrites the stored one.

Every cell time, *PWWFA* produces a new schedule. While *PWWFA* is working on this schedule (i.e. while several subschedulers run the waves through the matrix of elements), new requests may arrive, but they either fail to be picked up by the subscheduler working on the respective wave or they fail to be picked up early by a subscheduler that has several waves to process before completing its schedule. To further illustrate the first point, consider the following example in Figure 3.

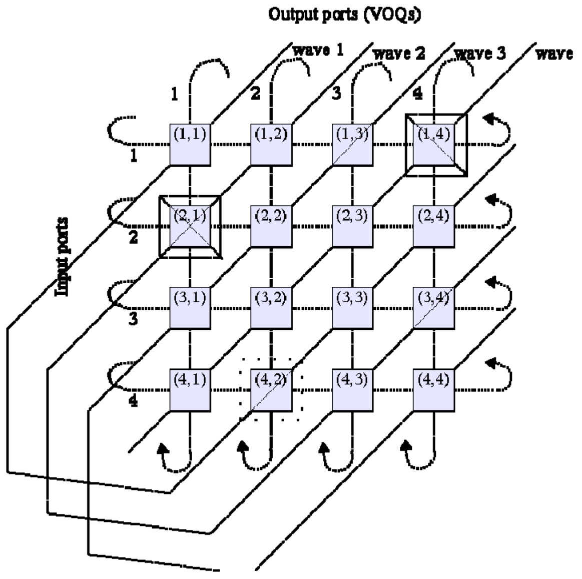


Figure 3. *PWWFA* arbiter for a 4x4 switch – *Fast Scheduler* example

Assume for simplicity that one subscheduler suffices and that initially, the matrix has requests in elements (1,4) and (2,1). This subscheduler processes wave 1 that consists of elements (1,1), (4,2), (3,3), and (2,4). None of these elements has requests, that is, $R_{sd}=0$ in these four elements, so obviously no grant is given.

Now, the subscheduler starts processing the second wave and grants a request in element (2,1). While this wave is being processed, a cell addressed to output port 2 arrives at input port 4. This increments the request counter in element (4,2), and sets register *R[4]=2*. However, this request *will not* be picked up by the subscheduler because this element belongs to wave 1 that has already been considered in this scheduling cycle. In fact, the request at element (4,2) will not be picked up by the scheduler until the next scheduling cycle.

This is where the *FS* helps. Once the schedule that includes grants for input ports 1 and 2 (for outputs 4 and 1, respectively) is ready, the *FS* will augment it with a grant for the request in register *R[4]*.

Note that it is possible that a schedule created by *PWWFA-FS* will ask for a cell that has already been sent out. Consider the following scenario.

A single cell arrives at input port 1, and it is addressed to output port 3 (it is stored in VOQ_{13}). The switch is otherwise empty. This arrival increments counter R_{13} which now equals 1, and sets register *R[1]=3* in the *FS*. Before *PWWFA* schedules this request, an empty schedule has been passed from *PWWFA* to the *FS*, which schedules a cell from VOQ_{13} and resets *R[1]*. In the implementation shown in Figure 2, there is no feedback from the *FS* to *PWWFA* so *PWWFA*, which can access only counters R_{sd} in the matrix, does not know that the cell has already been scheduled by the *FS*. In this case, *PWWFA* schedules a cell that in fact has already been removed from VOQ_{13}. Obviously, this takes away bandwidth as *PWWFA* could have instead scheduled another cell not scheduled otherwise.

Without any feedback from the *FS* to *PWWFA*, every request granted by the *FS* will be also granted by *PWWFA*. While this effect may appear to be detrimental to the overall performance of the scheme, two mechanisms counterbalance its impact. First, the *FS* comes into effect mostly when the load is light and bandwidth is abundant such that a loss in bandwidth can be tolerated to provide lower latency. Second, under heavy load, the *FS* rarely has a chance to augment schedules created by *PWWFA* so the scenario described above is rare.

Alternative implementations could synchronize the operation of the *FS* and *PWWFA* to avoid the above scenario. For example, requests granted by the *FS* could be removed from the *PWWFA* matrix by subsequently decrementing the appropriate counters R_{sd} or they could first be sent to the *FS* and forwarded to *PWWFA* only if rejected by the *FS*. Both schemes have advantages and drawbacks over the original scheme. While they avoid duplicate grants, the first scheme requires a feedback path from the *FS* to *PWWFA* and additional logic to decrement the counters R_{sd} and prevent cases of $R_{sd}<0$; the second scheme adds the latency of the *FS* to requests processed by *PWWFA*.

In the remainder of this paper, we will focus on the original scheme keeping the two alternatives in mind for further analysis in future publications.

The next section describes the *Fast Scheduler* in detail, emphasizing its practicality and ease of implementation. In a nutshell, the *FS* converts the schedule into bit vector representations that are further manipulated to match requests in registers *R[s]* in addition to requests scheduled by *PWWFA*. Some of these operations can be performed concurrently, and efficient bit vector manipulations assure their fast execution.

3.1 *PWWFA-FS* Step by Step

The execution of PWWFA-FS includes the following steps.

1. Building a PWWFA schedule and auxiliary bit vectors

As *PWWFA*'s subscheduler moves from wave to wave, a token of size $log_2(N)$ bits is passed from element to element in each row. If a request is granted, this token is set to the number of the output that was granted to a given input.

At the end of the scheduling cycle, the above tokens result in values *V[s]* stored in the matrix elements corresponding to the last wave of the scheduling cycle. *V[s]=d* is a binary representation of the output d matched with input *s* with *V[s]=0* if *s* was not matched and $0<s\leq N$. Note that *V* defined as a vector consisting of *V[1],V[2],...,V[N]* is in fact a complete schedule created by *PWWFA*, whose elements *V[s]* are distributed among *N* elements on a matrix diagonal.

The following auxiliary bit vectors can be derived from *V[1],V[2],...,V[N]*:

- $M=(M_N,...,M_1)$ is a bit vector that contains a 1 in position *s* if input *s* was matched by *PWWFA*, and a 0 otherwise, i.e. $M_s=1$ if $V[s]\neq 0$, and $M_s=0$ if $V[s]=0$
- $U=(U_N,...,U_1)$ is a bit vector that contains a 1 in position *d* if output *d* was matched by *PWWFA*, and a 0 otherwise, i.e. $U_d=1$ if $\exists\ s{:}V[s]=d$, and $U_d=0$ otherwise

Example. Consider the 4x4 matrix from Figure 3 with granted requests in elements (1,4) and (2,1). Table 2 shows inputs *s*, outputs *d*, *V[s]*, M_s, and U_d at the end of the scheduling cycle after the final wave, i.e. wave 4, has been processed. Note that inputs 3 and 4 and outputs 2 and 3 have not been matched.

Table 2. Contents of the elements on the final wave after completion of the *PWWFA* algorithm

Diagonal element	Input s	Output d	Schedule		
			V[s]	M_S	U_d
(1,4)	1	4	4	1	1
(2,3)	2	3	1	1	0
(3,2)	3	2	0	0	0
(4,1)	4	1	0	0	1

Deriving *M* from *V* is trivial since each M_s can simply be computed by comparing *V[s]* to zero. On the other hand, deriving *U* from *V* would require a search through the schedule for each output *d* to find a matching *s* such that *V[s]=d*. However, *M* and *U* can be more easily obtained during operation of *PWWFA*. After scheduling a wave, each element *(s,d)* of the wave signals adjacent elements if input *s* or output *d* have been matched. More specifically, element *(s,d)* signals the element to the right, i.e. (*(s mod N)+1,d*), whether input *s* has been matched, and it signals the element downwards, i.e. (*s, (d mod N)+1*), whether output *d* has been matched. By recording this information during operation of *PWWFA*, each element *(s,d)* of the final wave will not only store *V[s]*, but also M_s and U_d.

2. Sending the results to the *Fast Scheduler*

Each diagonal element of the final wave sends input number *s*, matched output *V[s]*, and bits M_s and U_d to the *FS*.

3. Pruning and matching requests in registers *R[s]*

The *FS* knows which *inputs* are eligible to be matched as identified by 0s in bit vector *M*. It further knows which *outputs* are eligible to be matched as identified by 0s in bit vector *U*. However, there may be inputs that request the same output, i.e. there may be pairs of inputs s_i and s_j with $R[s_i]=R[s_j]$. We propose two schemes for pruning these requests and finding valid matches between unmatched inputs and outputs. We will describe these schemes and their implementations in detail in Section 4. Both schemes require a conversion of binary requests *R[s]* into corresponding one-hot representations RF_{sd} with $RF_{sd}=1$ if *R[s]=d*, and $RF_{sd}=0$ otherwise. That is, each input *s* presents a bit vector $RF_s=(RF_{sN},...,RF_{s1})$ to the *FS*. Based on *R*, *RF*, *M*, and *U*, the *FS* computes a matching *W* in binary representation, where for each input *s* *W[s]=d* if *s* was matched with an output *d* and *W[s]*=0 otherwise.

4. Updating the schedule *V*

The *FS* produces a conflict-free matching *W* between inputs and outputs not matched by *PWWFA*, which means that the sets of inputs and the sets of outputs matched in *V* and *W* are disjoint. Therefore, updating the original schedule *V* with the augmentation *W* is trivial. For each input *s*, we define the final schedule *X[s]* as *X[s]=V[s]* if $V[s]\neq 0$ and *X[s]=W[s]* otherwise. This operation can simply be implemented as a bit-wise OR of *V* and *W*.

Example. Let the *FS* addition to the schedule be *W[3] = 2*. The complete, augmented schedule is now *X=(1,4), (2,1), (3,2), (4,0)*.

4. PRUNING MULTIPLE REQUESTS FOR THE SAME OUTPUT

Conflicts among the requests in registers *R[s]* have to be resolved quickly in the process of augmenting the schedule produced by *PWWFA*. In this context, the primary goal of the *FS* is low scheduling latency. The scheduling latency accounts for a large portion of the overall forwarding delay specifically under light load. Under heavy load, queuing delays tend to dominate. Therefore, the *FS* particularly has to be simple enough to make a scheduling decision in significantly less time than *PWWFA*. Ideally, the *FS* is simple enough to be implemented in hardware that can make a scheduling decision within a cell time. The scheduling decisions neither have to be optimal nor work-conserving and can exploit the assumption that the load is low. Also, fairness between the inputs is a secondary consideration.

In the following, we describe two possible schemes. The first scheme removes *all* requests that cause conflicts, while the second scheme removes *all but one* request for the same output.

4.1 *"Remove All"* Scheme

In the "*Remove All"* scheme, the *FS* augments a schedule calculated by *PWWFA* by granting requests to those outputs that have exactly one request and were not scheduled by *PWWFA*. If an output *d* receives exactly one request, i.e. there is a unique input *s* with *R[s]=d*, the request is granted and input *s* can send its cell immediately. If there are multiple requests for an output, all of the requests are rejected and have to wait to be scheduled by *PWWFA*.

While this scheme may leave outputs unmatched that could be matched, it is easy to implement and performs well under light load as we will show in Section 6.

Figure 4 shows a hardware implementation of the "*Remove All*" scheduler for one output *d* of a 4x4-switch. The requests RF_{1d}, RF_{2d}, RF_{3d}, and RF_{4d} from inputs 1, 2, 3, and 4 to output *d* are first masked with bit vector *M* to eliminate those inputs that have already been scheduled by *PWWFA*. The remaining requests are processed in two binary tree structures. The first tree, consisting of OR-gates, determines if there is at least one request and the second tree, consisting of one layer of AND-gates and a subtree of OR-gates, determines if there is at most one request. If both conditions are satisfied and the output has not been matched by *PWWFA* as indicated by bit U_d, the scheduler grants the output *d* to the input *s* that had the request. Since a grant implies that there was one unique input with a request, the grant signal is simply broadcast to all inputs. The inputs mask the grant signal with their request signal and the unique input *s* that had the request receives a grant G_{sd}. As there is one "*Remove All*" scheduler for each output, each input receives *N* grant signals G_{s1} through G_{sN}, of which only one can be set. Each input *s* merges G_{s1} through G_{sN} in a tree of OR-gates or with a wired OR into G_s= G_{s1} OR G_{s2} ... OR G_{sN}. Input *s* then sets *W[s]=R[s]* if $G_s=1$, and *W[s]=0* if $G_s=0$.

Note that the "*Remove All*" scheduler provides fairness among all inputs since either all requests are rejected or one is granted, but no request is ever chosen over another one. The resolution of conflicts is deferred to *PWWFA*.

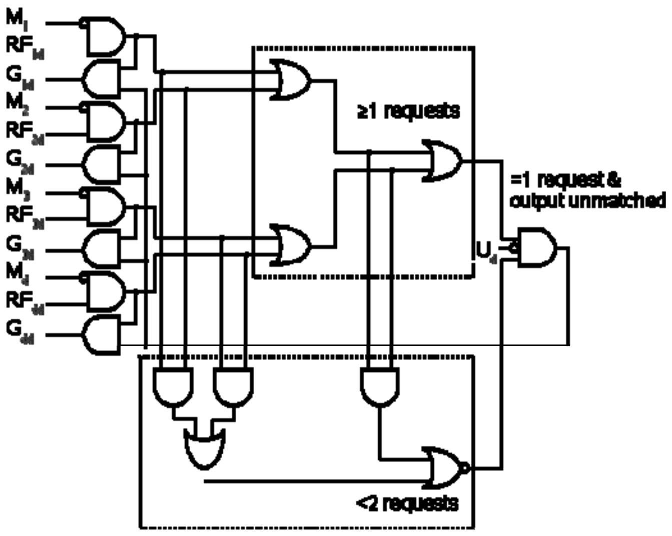


Figure 4. Schematics of the *"Remove All"* scheme

4.1.1 *Complexity of the "Remove All" Scheme*

The "*Remove All*" scheme as shown in Figure 4 is simple to implement in hardware and easy to analyze in size and delay. To estimate the size of the circuit, we first calculate the number of 2-input gates of an arbiter for one output port in an *N*-port switch. For a switch with *N* ports, there are *N* AND-gates to mask the requests *RF* with the vector of matched inputs *M*. The tree structure that detects the presence of at least one request consists of *N-1* OR-gates. Similarly, the structure to test for two requests to the same output requires *N-1* AND-gates and a subtree of *N-2* OR-gates. Adding two AND-gates to merge the outputs of the two tree structures and *N* AND-gates for masking the global grant signal, the scheduler requires *5N-2* 2-input gates for each output port arbiter. Assuming a tree of *N-1* OR-gates for each input s to merge G_{s1} through G_{sN} the total number of 2-input gates for an *N*-output scheduler is $6N^2-3N$. Looking at a 256x256-port switch as an example, the scheduler would require a reasonable number of 392,448 gates.

While the size of the scheduler grows quadratically with the number of ports, the use of tree structures leads to only a logarithmic growth in logic delay. The longest path includes one request-masking AND-gate (M_s AND R_{sd}), the "<2 requests" tree, the merging AND-gate, the grant-masking AND-gate, and the grant-merging OR-tree. It passes through $2log_2(N)+4$ gates. While we provide delay estimates based on the number of logic levels, we acknowledge that the performance of the "*Remove All*" scheme for large port counts will further depend on data locality, fanouts, and associated wire delay. However, compared to the scheduling latency of *PWWFA*, which grows linearly with the port count *N*, the delay of the "*Remove All*" scheme grows much slower. This way, scheduling decisions are made significantly faster especially for a large number of ports.

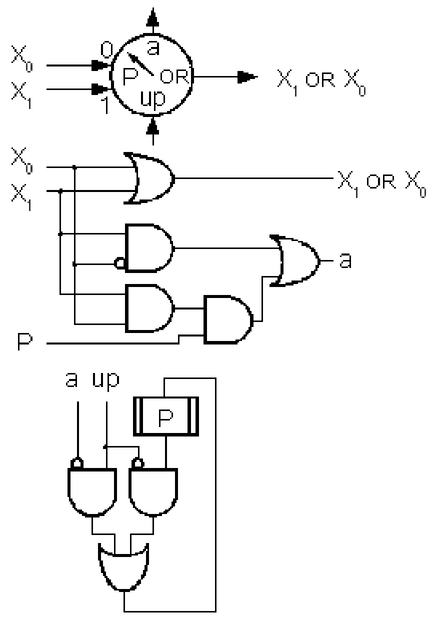
4.2 *"Leave One"* Scheme

In the "*Leave One*" scheme, the *FS* considers multiple requests for a given output and accepts one of the conflicting requests. The arbitration of the requests is performed in a tree structure of binary arbiters.

A binary arbiter, its corresponding truth table, and its gate-level implementation are shown in Figure 5. The arbiter has three inputs: two request inputs X_0 and X_1, and an update input *up*. It further has internal state in the form of priority bit *P* represented in Figure 5 by an arrow pointing to the request input of higher priority. Bits X_0 and X_1 represent the inputs that compete for grants, and priority *P* is used to resolve conflicts when both inputs request the output simultaneously, i.e. when they compete for a grant from the same output. The update input *up* controls the updating of priority register *P*.

One of the outputs of the binary arbiter is the logical OR of bits X_0 and X_1. The other output, referred to as *address bit a*, indicates which request has been granted. As multiple requests can compete for the same output, we need this bit at every level of the tree in order to keep track of the origin of the granted request. As the truth table in Figure 5 shows, when both inputs X_0 and X_1 request the same output, the selection of the request to be granted depends on the value of the priority bit *P*. To provide fairness, the internal priority bit *P* can be updated by asserting *up* as will be further explained in the context of Figure 6.

Figure 6 shows an implementation of the "*Leave One*" scheme for one output *d* of a 4x4-switch. As in the "*Remove All*" scheme, the requests RF_{1d}, RF_{2d}, RF_{3d}, and RF_{4d} are first masked with bit vector *M* to eliminate requests from inputs already scheduled by *PWWFA*. The remaining requests are then forwarded to a tree of binary arbiters. Each binary arbiter determines if there was a request on at least one input as indicated by output *OR*. Depending on priority *P*, the arbiter further determines which input should receive the grant as indicated by address output *a*.



X_1	X_0	P	X_1 OR X_0	a
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

up	a	P_{old}	P_{new}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 5. Binary arbiter with truth table and gate-level implementation

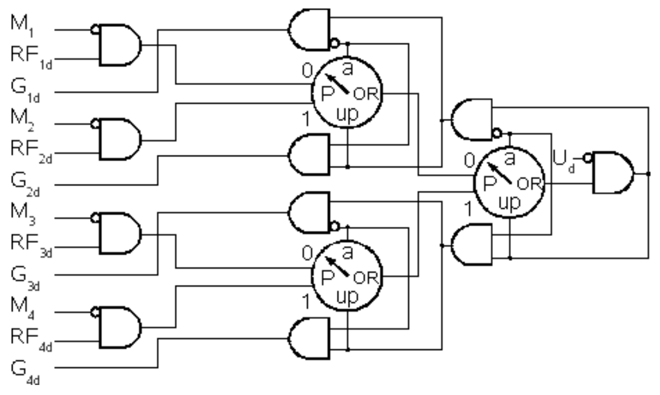


Figure 6. Schematics of the *"Leave One"* scheme

At the root arbiter on the right, the output of the OR-tree indicates if there was a request from at least one input. This signal is masked with bit U_d to ensure that output d did not receive a grant from *PWWFA*. If there is at least one request and output d was not scheduled by *PWWFA*, the arbiter sends a grant signal through an inverse tree of AND-gates to exactly one input s. At each binary arbiter, or node of the tree, the grant signal is either forwarded to the upper or lower branch of the tree as determined by the local address bit a. The arbiter thereby constructs a path from the root of the tree to the granted input s, which receives a grant signal G_{sd}. Similar to the "*Remove All*" scheme, there is one arbiter as shown in Figure 6 for each output d such that each input s receives N grant signals G_{s1} through G_{sN}. Since inputs s submit only one request $R[s]$ to one output, only one of the grant signals can be set and they can be merged in a tree of OR-gates or with a wired OR into $G_s = G_{s1}$ OR G_{s2} ... OR G_{sN}. Input s then sets $W[s]=R[s]$ if $G_s=1$, and $W[s]=0$ if $G_s=0$.

To provide some level of fairness between competing inputs, the "*Leave One*" arbiter shown in Figure 6 updates the priority bits of all binary arbiters on the path from the root arbiter on the right to the granted input on the left after each arbitration cycle that resulted in a grant. The priority bits P on the path to the granted input then point away from the granted input such that this input has the lowest priority of all inputs in the next cell slot. In an alternative implementation, random values could be assigned to the priority bits P of all binary arbiters after each arbitration cycle that resulted in a grant.

While both schemes with deterministic and random assignment of priorities P are easy to implement and work well for most traffic patterns, the tree structure of the arbiter can lead to unfairness under specific traffic patterns. If many requests frequently originate from the same subtree of the arbiter and few requests from another subtree, the latter ones receive grants disproportionately more often. For example, if in Figure 6 requests RF_{1d}, RF_{2d}, and RF_{4d} were always set, RF_{4d} would be granted half of the time and RF_{1d} and RF_{2d} one quarter of the time each. However, we expect this unfairness to have little impact under low load. Under high load, most requests will be processed by *PWWFA* hiding the unfairness of a "*Leave One*" *Fast Scheduler*.

4.2.1 Complexity of the "Leave One" Scheme

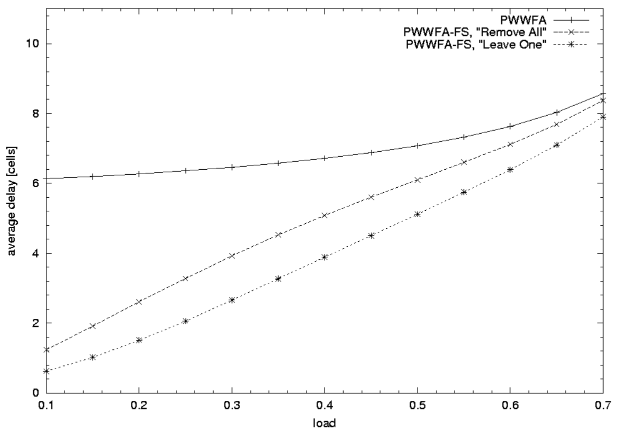
To compare the "*Leave One*" scheme to the "*Remove All*" scheme, we estimate the size and logic delay of the circuit based on 2-input gates. For a switch with N ports, there are N arbiters, one for each output port. Each arbiter uses N AND-gates to mask the requests RF with the vector of matched inputs M. The tree of binary arbiters consists of N-1 binary arbiters with 8 gates and one register bit each and the inverse AND-tree (including the masking gate for U_d) has $2N$-1 AND-gates. The N inputs each use a tree of N-1 OR-gates to merge grant signals G_{s1} through G_{sN}. Summed up, the total number of gates for an N-output scheduler is $12N^2$-$10N$ and the number of register bits is N^2-N. Looking at an example of a 256x256 switch, the "Leave One" scheduler includes 256 8-level trees with 255 binary arbiters each and a total number of 783,872 2-input gates and 65,280 register bits.

The logic delay is mainly determined by the delay of the three tree structures. The longest path includes the masking AND-gates, the OR-tree in the binary arbiters (except the root arbiter), three gates to generate address bit a in the root arbiter, the inverse tree of AND-gates, and the grant-merging OR-tree, for a total logic delay equivalent to $3log_2(N)+3$ gates. It is worth noting that the largest fanout of the "*Leave One*" scheduler is 3 gates independent of the number of ports N. This makes the "*Leave One*" scheduler amenable to high-speed hardware implementations even for large configurations.

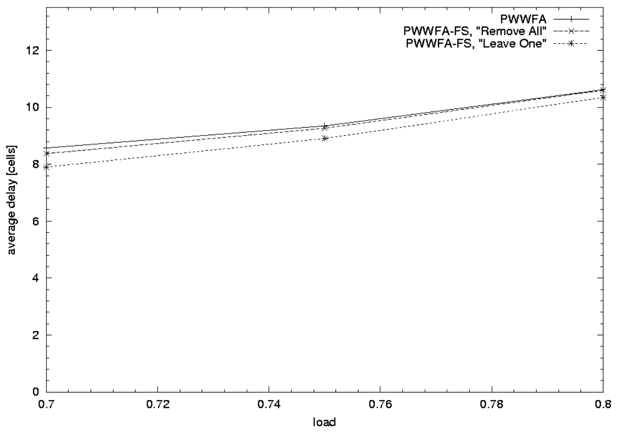
5. COMPLEXITY OF THE FAST SCHEDULER

As the sections above have shown, the implementation of both the "*Remove All*" and the "*Leave One*" scheme are simple. The steps of the main algorithm described in Section 3 can be executed using elementary logical operations on bit vectors. In addition, most operations for pruning and matching as described in Section 4 rely on tree structures that grow logarithmically in delay with the number of ports.

Therefore, in the simulation study below, we assume a conservative 20ns delay for the *Fast Scheduler*. Obtaining a more precise value of this delay is beyond the scope of this paper, as it depends on many factors such as chip technology and concrete circuit layout. We plan to address this aspect in our future work.



(A)



(B)

Figure 7. Average delay vs. load for Bernoulli traffic under light (A), and heavy load (B), *N=256*

6. PERFORMANCE OF *PWWFA-FS*

In this section, we compare the average delay (including queuing delay) of the following schemes: (1) our original *PWWFA* as presented in [9], (2) the proposed *PWWFA-FS* scheduler in which all conflicting requests for the same output port are rejected (*PWWFA-FS "Remove All"*), and (3) *PWWFA-FS* in which one of the conflicting requests is considered (*PWWFA-FS "Leave One"*).

Simulation results were obtained with a 95% confidence interval, not greater than 1% for the average delay. As in [9], we assume a switch with *N=256* ports, cell size *L=128*B, link rate *C=10*Gbps and a *Fast Scheduler* delay of 20ns. The number of *PWWFA* subschedulers is *S=10* (see [9] for more detail).

We used Bernoulli distributed traffic, and bursty, on-off traffic in which "on" and "off" periods are exponentially distributed. The "on" period has a mean length of 10 cells, and the mean length of the "off" period is adjusted depending on load λ as follows: $t_{off}=t_{on}\times(1-\lambda)/\lambda$. Results for on-off traffic are shown in the Appendix.

Figure 7A and Figure 7B show results for a 256-port scheduler with the delay given in cell times. Figure 7A focuses on light loads under which our new schemes perform best, and Figure 7B shows average delays vs. load for higher loads.

The differences between our original *PWWFA* and the new *PWWFA-FS* schemes are significant. For the lowest load of 0.1, *PWWFA* has a latency of 6.13 cell slots (equivalent to a delay of 628ns[1]), while *PWWFA-FS "Remove All"* has a latency of 1.24 cell slots (127ns), which is nearly 5 times less than the latency of *PWWFA*. The *PWWFA-FS "Leave One"* scheduler lowers the latency further to 0.62 cell slots (64ns) under a load of 0.1, which is nearly 10 times less than the *PWWFA* latency.

All curves converge with increasing loads, decreasing the benefits of the *Fast Scheduler*. Notably, even at 0.5-0.7 loads, the advantages of *Fast Scheduler* schemes are still visible.

The better performance of these schemes under light loads is understandable – they perform best when a schedule produced by *PWWFA* has many unmatched input/output ports. This is when the *Fast Scheduler* has the highest chance to add new matches. Confirming this effect in simulation, Figure 8 shows the fraction of a schedule filled by the *Fast Scheduler*. Not surprisingly, this fraction is highest under light loads.

Figure 7B shows results for higher loads. Even under these loads, the new schemes perform better than the original *PWWFA* but the differences are far less significant than for lower loads.

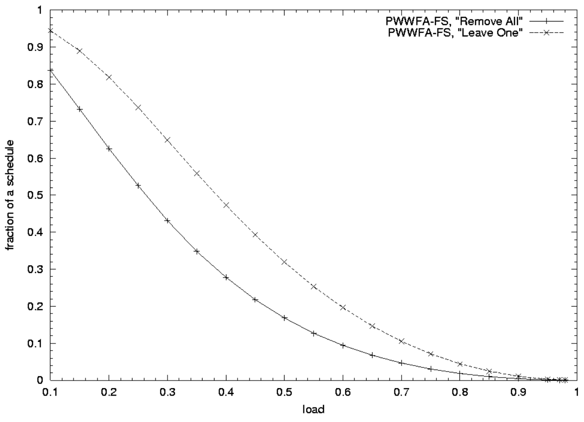
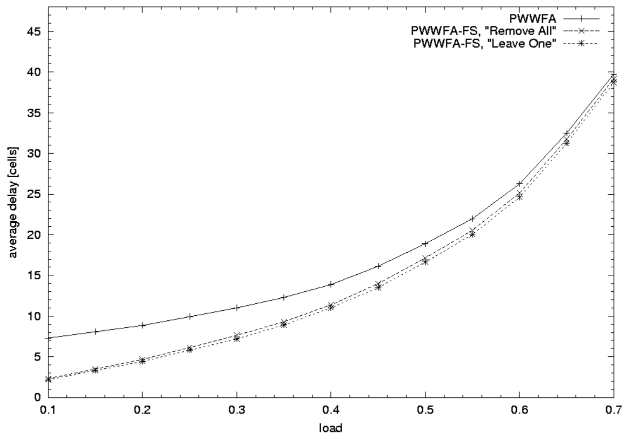


Figure 8. Fraction of the schedule filled up by the *Fast Scheduler* vs. load for *N=256* for Bernoulli traffic

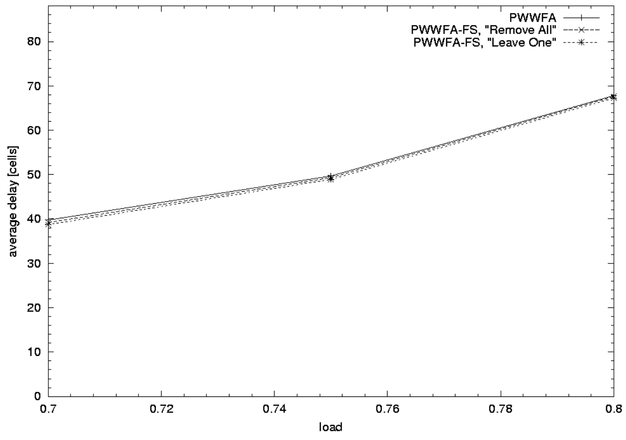
We also ran simulations for a considerably smaller switch with 32 ports, cell size *L=64*B, port rate of *10*Gbps, and *S=3* subschedulers. These simulations show similar relations, although the improvements given by *Fast Scheduler* schemes are less pronounced. For example, under a load of 0.1, the latency of *PWWFA* is 2.13 cell slots, which is 109ns (cells in 32-port

[1] 512ns of this delay comes from waves traversing the *PWWFA* matrix.

simulations are 64B long). The latency of *PWWFA-FS "Remove All"* is 1.2 cell slots (61ns), which gives a 1.7-fold improvement over the *PWWFA* latencies, and the latency of *PWWFA-FS "Leave One"* is 1.1 cell slots (56ns), which gives a nearly 2-fold improvement.



(A)



(B)

Figure 9. Average delay vs. load for on-off traffic under light (A), and heavy load (B), *N=256*

7. FUTURE WORK AND CONCLUSIONS

We presented the Parallel Wrapped Wave Front Arbiter with Fast Scheduler (*PWWFA-FS*) in which we extended our original *PWWFA* scheme. This centralized scheduler assures low latencies in large, terabit-per-second switches with hundreds of ports. We are not aware of any schedulers capable of achieving such low latencies for large-scale switches.

We are currently working on the implementation details of the *PWWFA-FS* algorithm in the context of a large switch. We plan to incorporate it in a switch we are currently developing [4] that is based on Proximity Communication [3]. This work will also shed more light on the actual implementation complexity of and delays introduced by *PWWFA-FS*. While we limit the discussion in this paper to augmenting a schedule produced by *PWWFA*, the *Fast Scheduler* could likewise augment schedules produced by other algorithms such as the ones proposed in [7] and [8].

We also want to look closer at the unfairness of the *PWWFA* algorithm that comes from the unfairness of the original *WWFA* ([10], [11]). We plan to run simulations for other traffic patterns whose purpose will be to obtain measurements that will show quantitatively what impact the unfairness has on delays. Also, we want to investigate the impact of bandwidth loss due to the lack of feedback between the *Fast Scheduler* and *PWWFA*, and analyze the two alternative schemes that avoid a loss of bandwidth.

8. ACKNOWLEDGEMENTS

The authors wish to acknowledge the anonymous reviewers, and Nikos Chrysos for their comments.

9. APPENDIX

Figure 9A and Figure 9B show results for a 256-port scheduler and on-off traffic. Figure 9A shows average delays vs. load under lighter loads, and Figure 9B focuses on heavier loads. We observe a 3.3-fold decrease of latency for light loads.

10. REFERENCES

[1] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High Speed Switch Scheduling for Local Area Networks", *ACM Trans. Comput. Syst.*, vol. 11, no. 4, pp. 319-352, Nov. 1993

[2] H. J. Chao and J.-S.Park, "Centralized Contention Resolution Schemes for a Large-Capacity Optical ATM switch", *Proc. IEEE ATM Workshop'97*, Fairfax, VA, May 1998

[3] R. Drost, R. D. Hopkins, and I. Sutherland, "Proximity Communication", I*EEE Custom Integrated Circuits Conference*, pp. 469-472, Sep. 2003

[4] H. Eberle, A. Chow, B. Coates, J. Cunningham, R. Drost, J. Ebergen, S. Fairbanks, J. Gainsley, N. Gura, R. Ho, D. Hopkins, A. Krishnamoorthy, J. Lexau, W. Olesinski, J. Schauer, and T. Ono, "Multi-terabit Switch Fabrics Enabled by Proximity Communication", *Symposium on High Performance Chips (Hot Chips'07)*, Stanford, CA, August 19-21, 2007

[5] I. Elhanany, and X. Li, "A Scalable Frame-based Multi-crosspoint Packet Switching Architecture", *High Performance Switching and Routing Conference (HPSR'07)*, Brooklyn, New York, May 30-June 1, 2007

[6] N. McKeown, "The iSlip Scheduling Algorithm for Input-queued Switches", *IEEE/ACM Transaction on Networking*, vol. 7, no. 2, Apr. 1993

[7] C. Minkenberg, I. Iliadis and F. Abel, "Low-latency Pipelined Crossbar Arbitration", *IEEE Global Telecommunications Conference 2004* (*GLOBECOM '04*), vol. 2, pp. 1174-1179, Nov. 2004

[8] E. Oki, R. Rojas-Cessa, and H.J. Chao, "A Pipeline-Based Maximal-Sized Matching Scheme for High-Speed Input-Buffered Switches", *IEICE Transactions on Communications*, vol. E85-B, no. 7, pp. 1302-1311, July 2002

[9] W. Olesinski, H. Eberle, and N. Gura, "PWWFA: Parallel Wave Front Arbiter for Large Switches", *High Performance Switching and Routing Conference (HPSR'07)*, Brooklyn, NY, May 30-June 1, 2007.

[10] Y. Tamir and H. C. Chi, “Symmetric Crossbar Arbiters for VLSI Communication Switches”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, issue 1, pp. 13 – 27, Jan. 1993

[11] Y. Tian, X.Tu , L. Wen, K. Wang, and Y. Liu, “PPAwFE: a Novel High Speed Crossbar Scheduling Algorithm”, *Proceedings of the 2005 International Conference on Communications, Circuits and Systems*, vol.1, pp. 673-677, May 2005