Scheduling in Non-Blocking Buffered Three-Stage Switching Fabrics

Nikos Chrysos and Manolis Katevenis[‡]

Foundation for Research and Technology - Hellas (FORTH), member of HiPEAC

***Abstract*— Three-stage non-blocking switching fabrics are the next step in scaling current crossbar switches to many hundreds or few thousands of ports. Congestion (output contention) management is the central open problem –without it, performance suffers heavily under real-world traffic patterns. Centralized schedulers for bufferless crossbars manage output contention but are not scalable to high valencies and to multi-stage fabrics. Distributed scheduling, as in buffered crossbars, is scalable but has never been scaled beyond crossbars. We combine ideas from centralized and from distributed schedulers, from request-grant protocols, and from credit-based flow control, to propose a novel, practical architecture for scheduling in non-blocking buffered switching fabrics. The new architecture relies on multiple, independent, single-resource schedulers, operating in a pipeline. It: *(i)* does not need internal speedup; *(ii)* directly operates on variable-size packets or multi-packet segments; *(iii)* isolates well-behaved from congested flows; *(iv)* provides delays that successfully compete against output queueing; *(v)* provides 95% or better throughput under unbalanced traffic; *(vi)* provides weighted max-min fairness; *(vii)* resequences cells or segments using very small buffers; *(viii)* can be realistically implemented for a 1024×1024 reference fabric made out of 32×32 buffered crossbar switch elements at 10 Gbps line rate. This paper carefully studies the many intricacies of the problem and the solution, discusses implementation, and provides performance simulation results.**

1 . Introduction

Switches are increasingly used to build the core of routers, cluster and server interconnects, other bus-replacement devices, etc. The desire for scalable systems implies a demand for switches with ever-increasing valency (port counts). Beyond 32 or 64 ports, single-stage crossbar switches are quite expensive, and *multi-stage interconnection networks (switching fabrics)* become preferable; they are made of smaller-valency switching elements, where each such element is usually a crossbar. It has been a longstanding objective of designers to come up with an economic interconnection architecture, scaling to large port-counts, and achieving sophisticated quality-of-service (QoS) guarantees under unfavorable traffic patterns. This paper addresses that challenge.

The performance of switching fabrics is often severely hurt by inappropriate decisions on how to share scarce resources. *Output contention* is a primary source of such difficulties: input ports, unaware of each other's decisions, may inject traffic for specific outputs that exceeds those outputs' capacities. The excess packets must either be dropped –thus leading to poor performance– or must wait in buffers; buffers filled in this way may prevent other packets from moving toward their destinations, again leading to poor performance. Tolerating output contention in the short term, and coordinating the decisions of input ports so as to avoid output contention in the long run is a complex *distributed scheduling* problem; flow control and congestion management are aspects of that endeavor. This paper contributes toward solving that problem.

Switching fabrics may be *bufferless* or *buffered*. Bufferless fabrics merely steer traffic, without being able to delay some of it in favor of other. Such fabrics cannot tolerate any output contention (or contention for internal links), thus they impose very stringent requirements on the scheduling subsystem. Buffered fabrics, on the other hand, contain some internal temporary storage so as to tolerate contention up to a certain extent. Buffered fabrics are clearly preferable, and modern integrated circuit technology makes them feasible. This paper assumes such *buffered fabrics*, and is concerned with how to reduce buffer requirements and how to control the use of limited buffer space.

Buffers in fabrics are usually small [1], so as to avoid off-chip memory at the switching elements and limit delays through the fabric. In order for the small buffers not to overflow, *backpressure* is used. Indiscriminate backpressure stops all flows sharing a buffer when that buffer fills up; it leads to poor performance due to buffer hogging –a phenomenon with effects similar to head-of-line blocking. Per-flow buffer reservation and per-flow backpressure signaling overcome these shortcomings, but become expensive with increasing number of flows. Per-destination flow merging [2] alleviates this cost. One practical compromise is to dynamically share the available buffer space among flows destined to multiple (as many as possible) distinct output ports, as in the ATLAS I chip [3]. A related, improved method is to dynamically detect congestion trees, allocate "set-aside queues (SAQ)" to them, and use per-SAQ backpressure [4].

This paper proposes and evaluates an alternative, novel scheduling, congestion management, and flow control architecture: when heavy traffic is detected, input ports have to first request and be granted permission before they can send any further packets. Requests are routed to and grants are generated by a scheduling subsystem. This subsystem, which can be central or distributed, consists of independent, per-output and per-

[‡] The authors are also with the Dept. of Computer Science, University of Crete, Heraklion, Crete, Greece.

input, single-resource schedulers, operating in parallel. The architecture has conceptual analogies to scheduling in buffered crossbars (combined input-crosspoint queueing - CICQ) [5] [6]. Compared to the alternatives listed above, the new method: *(i)* operates robustly under all traffic patterns, not just under "typical" traffic; *(ii)* economizes on buffer space; and *(iii)* applies to scalable *non-blocking* fabrics that employ multipath routing. Other previously proposed scheduling schemes for 3-stage non-blocking fabrics assumed one unbuffered stage, while our new architecture applies to fully buffered fabrics, thus yielding significantly higher performance. Further discussion on these comparisons appears in section 2 on related work. Before that, we describe our scheduling architecture and the switching fabric where it fits.

1.1 Non-Blocking Three-Stage Fabrics

Switching fabrics are said to present *internal blocking* when internal links do not suffice to route any combination of feasible I/O rates, hence, contention may appear on internal links as well, in addition to output ports. Otherwise, a fabric is called *non-blocking* when it can switch any set of flows that do not violate the input and output port capacity limits. Although internal blocking clearly restricts performance, most commercial products belong to this first category, because a practical, robust, and economic architecture for non-blocking fabrics has not been discovered yet. However, neither has it been proven that such architectures do not exist. This paper contributes to the search for practical, robust, and economic non-blocking switching fabrics.

Low-cost practical non-blocking fabrics are made using *Clos* networks [7]; the basic topology is a three-stage fabric, while recursive application of the principle can yield 5- and more stage networks. One of the parameters of Clos networks, m/n, controls the *speed expansion* ratio –something analogous to the "internal speedup" used in combined input-output queueing (CIOQ) architectures: the number of middle-stage switches, m, may be greater than or equal to the number of input/output ports per first/third-stage switch, n. In this paper, we assume $m = n$, *i.e. no speedup* –the aggregate throughput of the middle stage is no higher than the aggregate throughput of the entire fabric. In this way, the fabrics considered here are the *lowest-cost* practical non-blocking fabrics, oftentimes also referred to as *Benes* fabrics [8].

In order for a Benes fabric to operate without internal blocking in a packet switching set-up, *multipath routing* (inverse multiplexing) must be used [9] [10]: each flow (as defined by an input-output port pair) is distributed among all middle-stage switches, in a way such as to equalize the rates of the resulting sub-flows. The middle-stage switches can be thought of as parallel slices of one, faster virtual switch, and inverse multiplexing performs load balancing among these slices. Such multipath routing introduces *out-of-order* packet arrivals at the output ports; we assume that egress linecards perform *packet resequencing*, so as to ensure in-order eventual packet delivery. Our scheduling system specificly *bounds* the extent of packet mis-ordering, thus also bounding the required size of reorder buffers, so that the latter can fit on-chip using modern IC technology for our 1024-port reference fabric –see section 4.2.

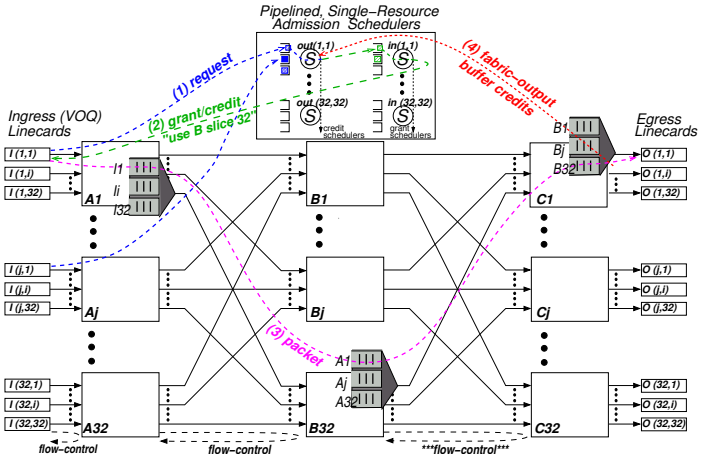

Fig. 1. a reference design for this paper.

1.2 Reference Design: a 1024×1024, 10 Tb/s Fabric

Although the architecture proposed and evaluated in this paper is quite general and applicable to many networks, our motivation for developing it, and our primary benchmark for it, is an example *next-generation fabric challenge*, that is realistic as a commercial product in the second half of this decade. This "reference design", shown in figure 1, is a 1024×1024 switching fabric (valency N=1024), made out of 96 single-chip 32×32 switching elements (3 stages of 32 switch chips of valency M=32 each), plus one (1) scheduler chip, shown in the top middle of the figure; linecards are not included in the chip counts. We consider that the line rate of each link is 10 Gbits/s or more, limited mostly by the power consumption of the switch chip I/O transceivers. We name the first, second, and third switch stages as A, B, and C respectively. Although this topology looks like current "byte-sliced" commercial switch products, where each cell is sliced into M subunits and concurrently routed through all B switches, our system is very different: cells (actually: variable-size segments) are routed intact (unsliced) through *one* of the B switches each, asynchronously with each other; resequencing is provided in the egress linecards.

We assume links carry *variable size* segments, each containing one or more variable-size packets or fragments thereof, as in [11], so as to eliminate padding overhead (if segments had fixed size) and reduce header and control overhead (by carrying multiple small packets inside a segment). Linecards are assumed to contain (large, off-chip) virtual-output queues (VOQ) in the ingress path, and (small, on-chip) resequencing and reassembly buffers in the egress path. No (large, off-chip) output queues are needed, since we do *not* need or use any internal speedup; in other words, this architecture has the same advantages as variable-packet-size buffered crossbars [12]. We assume that individual switch chips are buffered crossbars, like our recent chip design [12] which proved their feasibility in

the 2006-08 time frame for size 32×32, with few-Kilobyte buffers per crosspoint, at 10 Gb/s line rate. We chose buffered crossbars because of their simplicity, scheduling efficiency, and support for variable-size packets.

The scheduler chip is connected to each A switch via one link, and to each C switch via another link, for a total 64 links (not shown in the figure), just like each switch chip has 64 I/O links (32 in, 32 out). We chose the parameters of our reference design so that the scheduling subsystem can fit in a single chip, although this subsystem could also be distributed among multiple chips. To achieve a single-chip scheduler, we have to ensure that the aggregate throughput of its traffic does not exceed $1/M$ times the aggregate data throughput of the fabric, where M=32 is the switch valency, for the following reasons. Since the M switches in each fabric stage can pass the aggregate data throughput, it follows that the one scheduler chip can pass the aggregate control throughput, if the latter is $1/M$ times the former. The scheduler chip is connected to each A and C chip via one link; that link suffices to carry the control traffic that corresponds to the M data links of the switch chip, if control traffic is $1/M$ times the data traffic.

For these relations to hold for $M = 32$, we assume that the maximum-size segment is 64 Bytes or larger. Under heavy traffic, almost all segments are of maximum size, because they are allowed to carry multiple packets (or packet fragments) each. The control traffic, per segment, consists of a request (10 bits), a grant (10 bits), and a credit (5 bits) –see section 5.1. Hence, the data throughput, for a switch, per segment, is 1024 bits (512 entering, 512 exiting), while the control throughput, for the scheduler, per segment, is 25 bits (15 entering, 10 exiting); the resulting control-to-data ratio is $25/1024 \approx 1/41$ (bidirectional), or $15/512 \approx 1/34$ (entering) and $10/512 \approx 1/52$ (exiting).

1.3 Our Admission Scheduling Architecture

The basic idea of our scheduler is that, under heavy traffic, ingress ports have to request and be granted permission before they can send a segment to the fabric. The request-grant handshake incurs some delay, but that delay is in parallel with –hence masked by– the (VOQ) input-queueing delay. Only under light load would this extra delay be visible, but we assume that the request-grant protocol is not used for light-load flows. This point is further discussed in section 1.5 while the bulk of this paper concerns fabric operation under heavy load.

The request-grant protocol economizes on buffer space relative to per-flow buffer reservation and backpressure. Effectively, instead of first letting data occupy buffers and then scheduling among the flows to which these data belong ("corrective" congestion management), we schedule first among competing requests and then let into the fabric only the data that are known to be able to quickly get out it ("preventive" congestion management).

Schedulers for bufferless switches (usually crossbars) serve the same preventive function, but have a much harder time because they must enforce absolute admissibility of the traffic, per time-slot. Our scheduler only has to enforce admissibility over a longer time window, because the fabric contains internal buffers. This time window serves to mask the latency of the scheduling pipeline. At the same time, buffers allow some overprovisioning of traffic admissions. These excess admissions mask out scheduling inefficiencies (not being able to simultaneously match all inputs to all outputs). Thus, instead of using (expensive) internal throughput speedup, as in bufferless crossbars, we use admissions overprovisioning, which is almost for free given the low cost of buffer memory in modern chips. In essence, we achieve the scheduling efficiency of buffered crossbars, but at a cost that grows with[1] $O(N \cdot \sqrt{N})$ instead of $O(N^2)$.

Our admission method is realized by *independent* per-output and per-input single-resource schedulers, working in parallel (figure 1). Input requests specify the flow's output port, and are routed to the scheduler for that port. Requests are queued in front of the proper per-output (credit) scheduler; these queues often degenerate to mere counters. Each per-output scheduler generates grants after first allocating space in that output's buffer[2]. Grants can be generated according to a desired quality-of-service (QoS) policy, *e.g.* weighted round robin (WRR) / weighted fair queueing (WFQ). When the data that were granted eventually depart through that output, the scheduler is notified so as to re-allocate that buffer space. Thus, the rate of data departures indirectly regulates the rate of grant generation, while buffer size (minus control-protocol round-trip time (RTT)) determines the amount of admissions overprovisioning.

Multiple per-output schedulers may simultaneously generate grants for a same input port. A per-input scheduler serializes these in a desired order and forwards them to the input at a convenient rate. Per-output and per-input schedulers work in parallel, asynchronously from each other, in a pipeline fashion (they can even be in separate chips). As long as each single-resource scheduler maintains a decision rate of at least one result per segment time, admissions proceed at the proper rate.

The scheduling subsystem principles and operation are discussed in detail in section 3; the central scheduler organization and implementation is discussed in sections 4 and 5.

1.4 Contributions and Results Achieved

First, the paper presents a careful study of this novel scheduling architecture, its parameters, and its variants. We consider this class of architectures very interesting because they perform the function of bufferless-crossbar schedulers, but at the high efficiency of buffered-crossbar scheduling, while using significantly less buffer space than buffered crossbars, and while being scalable to high-valency fabrics.

[1] each switch has $\sqrt{N}$ ports, hence N crosspoint buffers; there are $\sqrt{N}$ switches per stage, hence $3 \cdot \sqrt{N}$ in the entire fabric. Thus, there are $3 \cdot N \cdot \sqrt{N}$ crosspoint buffers in the fabric.

[2] space should in general be reserved for intermediate-stage buffers as well; however, it turns out that, because the fabric is non-blocking, no serious harm results if such allocation is omitted –see section 3.3.

Second, the proposed architecture switches equally well fixed-size cells or *variable-size* (multi-packet) segments, because it only uses independent single-resource schedulers throughout. Thus, it retains the advantages of buffered crossbars: no padding overhead, thus no internal speedup needed, hence no (large, off-chip) output queues needed, either. Although no internal speedup is used, throughput under unbalanced traffic is very high. The simulations presented in this paper are mostly for fixed-size cells, in order to compare our results to alternative architectures that operate only on cells. However, we do present simulations showing smooth operation with variable-size segments.

Third, advanced QoS policies can be straightforwardly implemented in the proposed architecture. For example, we simulated the system using WRR/WFQ admission schedulers: under inadmissible traffic (persistently backlogged VOQs), the system distributes input and output port bandwidth in a *weighted max-min fair* manner; up to now, this had only been shown for single-stage buffered crossbars [6].

Fourth, we quantify buffer space requirements, using simulations. Interestingly, for good performance, a *single RTT-window buffer per-crosspoint* suffices, provided that this buffer size is at the same time large enough for *several segments (cells)* to fit in it (RTT is the control protocol round-trip time). As long as crosspoint buffers are larger than one RTT-window each, it appears that performance is sensitive to the number of segments per output port that can be pending inside the fabric at once. The (excellent) performance results listed in the next paragraph are achieved with crosspoint buffers on the order of ten (10) segments each, and assuming that the overall (control) RTT is equal to 10 segment times.

Finally, the new architecture achieves excellent performance *without any internal speedup*. Under uniformly-destined traffic, the system delivers 100% throughput, and delay within 1.5 times that of pure output queueing (OQ) for bursty traffic, and within 4 times that of OQ under smooth traffic[3]. Under unbalanced traffic, the simulated throughput exceeds 95%. Under hot-spot traffic, with *almost all* output ports being congested, the non-congested outputs experience negligible delay degradation (relative to uniform traffic); at the same time, the congested outputs are fully utilized (100% load). Compared to bufferless 3-stage Clos fabrics [13], our architecture performs much better, and, at the same time, uses a much simpler scheduler.

For the 1024×1024 reference design (section 1.2), these performance results can be achieved with 780 KBytes of total buffer memory per switch chip, assuming the overall control RTT can be kept below 600 ns, and assuming 64 Byte maximum segment size (hence, 12 segments per crosspoint buffer). Under the same assumptions, 25 KBytes of reorder buffer suffice in each egress linecard. Alternatively, if the control RTT is as high as 3.2 μs, if we increase maximum segment size to 256 Bytes (so as to reduce header overhead), and if we increase crosspoint buffer size to 16 segments = 4 KBytes (for even better performance), then buffer memory per switch chip will be 4 MBytes (feasible even today), and reorder buffer size will be 128 KBytes.

[3]results obtained using round-robin schedulers and fabric size up to 256×256 made of 16×16 switches.

Comparisons to related work appear in section 2. The operation of the scheduler is discussed in sections 3 and 4, and a practical implementation is presented in section 5. Performance simulation results are presented in section 6.

1.5 Eliminating Request-Grant Latency under light Load

The request-grant protocol adds a round-trip time (RTT) delay to the fabric response time. For heavily loaded flows this RTT delay is hidden within input queueing delay. For lightly loaded flows, it is desirable to avoid this extra delay in latency-sensitive applications, *e.g.* cluster/multiprocessor interconnects. We are currently studying such protocols, and we have promising preliminary simulation results. The basic idea of these protocols is as follows.

Every input is allowed to send a small number of segments without first requesting and receiving a grant. Once these segments have exited the fabric (as recognized by credits coming back), the input is allowed to send more segments without a grant. However, in order to send more segments before receiving credits for the previous ones, the input has to follow the normal request-grant protocol. Under light load, credits will have returned before the flow wishes to send new segments, thus allowing continued low-latency transmission. Under heavy load, the system operates as described in the rest of the paper. To guard against the case of several inputs by coincidence sending at about the same time "free" cells to a same output, thus creating a congestion tree, we consider having "free" cells and "request-grant" cells travel through separately reserved buffer space (and being resequenced in the egress linecard).

2 . Related Work

2.1 Per-Flow Buffers

In a previous study [2], we considered a buffered Benes fabric where congestion management was achieved using per-flow buffer reservation and per-flow backpressure signaling. To reduce the required buffer space from $O(N^2)$ down to $O(N)$ per switching element, where N is the fabric valency, we introduced per-destination flow merging. That system provides excellent performance. However, the required buffer space, at least in some stages, is $M \cdot N$, where M is the switch valency. In our reference design, M is relatively large in order to reduce the number of hops; thus, either the first or the third stage would need switches containing 32 K "RTT" windows each, which is rather large. Furthermore, the buffers in this space are accessed in ways that do not allow partitioning them for reduced throughput (e.g. per-crosspoint).

This paper addresses those practical problems: we only use O(M^2) buffer space per switch (only 1 K windows for the reference design), explicitely partitioned and managed per-crosspoint. This partitioning allows variable-size segment operation. Furthermore, the present architecture can provide

WMMF QoS, which would be quite difficult in [2], where merged-flow weight factors would have to be recomputed dynamically during system operation.

2.2 The Parallel Packet Switch (PPS)

The Parallel Packet Switch (PPS) [14] [15] is a three-stage fabric where the large (and expensive) buffers reside in the central-stage. First and third stage switches serve a single external port each. By increasing the number of central elements, k, the PPS can reduce the bandwidth of each individual memory module, or equivalently provide line-rate scalability. Essentially, the PPS operates like a very-high-throughput shared buffer, which is composed of k interleaved memory banks; one expensive and complex component of the design is how to manage the shared buffer data structures (queue pointers etc.) at the required very high rate, hence necessarily in a distributed fashion. The PPS provides port-rate scalability, but does not provide port-count (N) scalability. One could modify the PPS for port-count scalability, by modifying each first-stage element from a 1-to-k demultiplexor serving one fast input to an $M \times k$ switch serving M normal inputs; correspondingly, each third-stage element must be changed from a k-to-1 multiplexor to a $k \times M$ switch. However, this latter modification would require dealing with output contention on the new "subports", *i.e.* per-subport queues along the stages of the PPS. Effectively, then, this radically altered PPS would have to solve the same problems that this paper solves for the input-queued fabric.

2.3 Memory-Space-Memory Clos

Clos fabrics containing buffers in the first and last stages, but using bufferless middle stage, and having a central scheduler, have been implemented in the past [16] and further studied recently [13]. These schedulers are interesting but complex and expensive (they require two iSLIP-style exact matchings to be found, some of which among N ports, per cell-time). Like iSLIP, they can provide 100% throughput under uniform traffic, but performance suffers under non-uniform load patterns. In-order delivery results from (or is the reason for) the middle stage being bufferless. This paper demonstrates that the cost of allowing out-of-order traffic, and then reordering it in the egress linecard, is minimal. In return for this cost, the use of buffered crossbars in all stages of our architecture provides much better performance with a much more scalable scheduler.

2.4 Regional Explicit Congestion Notification (RECN)

A promising method to handle the congestion in multistage switches has recently been presented in [4]. A key point is that sharing a queue among multiple flows will not harm performance as long as the flows are not congested. Hence, [4] uses a single queue for all non-congested flows, and dynamically allocates a set-aside-queue (SAQs) per congestion tree, when the latter are detected. Congestion trees may be rooted at any output or internal fabric link, and their appearance is signaled upstream via "regional explicit congestion notification (RECN) messages. We consider [4] and our scheme as the two most promising architectures for congestion management in switching fabrics. Precisely comparing them to each other will take a lot of work, because the two systems are very different from each other, so the comparison results depend a lot on the relative settings of the many parameters that each system has.

Nevertheless, a few rough comparisons can be made here: *(i)* RECN saves the cost of the central scheduler, but at the expense of implementing the RECN and SAQ functionality (which includes a content-addressable memory) in every switch; *(ii)* under light load, RECN uses very little throughput for control messages; however, some amount of control throughput must be provisioned for, to be used in case of heavy load, and this may not differ much from control throughput in our system; *(iii)* RECN has not be studied for fabrics using *multipath* routing, which is a prerequisite for economical *non-blocking* fabrics, like our system does, hence it is not known whether and at what cost RECN applies to non-blocking fabrics; *(iv)* RECN works well when there are a few congestion trees in the network, but it is unknown how it would behave (and at what cost) otherwise, while our system operates robustly independent of the number of congested outputs (no internal links can ever be congested in our system); *(v)* contrary to our system, in RECN, during the delay time from congestion occurrence until SAQ setup, uncongested flows suffer from the presence of congested ones; *(vi)* RECN relies on local measurements to detect congestion; these measurements are performed on an output buffer; for reliable measurement (especially under bursty traffic or with internal speedup), that buffer cannot be too small; at the same time, RECN signaling delay translates into SAQ size; the sum of all these required buffer sizes may end up not being much smaller than what our system requires.

2.5 End-to-end Rate Regulation

Pappu, Turner, and Wong [17] [18] have studied a rate regulation method analogous to ours. Both systems regulate the injection of packets into a fabric so as to prevent the formation of saturation trees.

However, the Pappu system foresees a complex and lengthy communication and computation algorithm; to offset that cost, rate adjustments are made fairly infrequently (*e.g.*, every 100 μs). Such long adjustment periods *(i)* hurt the delay of new packets arriving at empty VOQs; and *(ii)* do not prevent buffer hogging and subsequent HOL blocking during transient phenomena in between adjustment times, when these buffers are not proportionally sized to the long adjustment period. Our scheme operates at a much faster control RTT, with much simpler algorithms, basically allocating buffer space, and only indirectly regulating flow rates. The result is low latencies and prevention of buffer hogging. Additionally, Pappu e.a. do not address the size of resequencing buffers, while we provide a quite low bound for that size.

3 . SCHEDULING THREE-STAGE NON-BLOCKING FABRICS

This section shows how to properly schedule, using independent and pipelined schedulers, a N-port non-blocking three-stage fabric, with as few as O(M) queues per $M \times M$ switch (M=$\sqrt{N}$). To that end, we combine ideas from bufferless and buffered fabrics. The first scheduler to be presented here is derived from first principles, and for that reason it is expensive and complicated; then we simplify it in sections 3.3 and 4.

3.1 Key Concepts

The first idea is to use an independent scheduler for each fabric buffer (this will later be relaxed). A packet (segment) will only be injected into the fabric after all schedulers for all buffers along its route have reserved space for the packet. First reserving then injecting trades latency (for the request-grant round-trip time (RTT)) for buffer space economy: buffers are only occupied by cells that are guaranteed to move forward, instead of being uselessly held by congested-flow cells, with backpressure protocols.

We start buffer-space reservations *from the last* (output) fabric stages, moving left (to the inputs), one stage at a time; this is precisely opposite to how cells progress under backpressure protocols. The direction chosen ensures that each reservation, when performed, is on behalf of a cell that is guaranteed not to block inside the buffer: buffer space has already been reserved for that cell in the next downstream buffer. Hence, cells will be allowed to move freely, without need for any backpressure to ever hold them back, and without danger of any buffer overflowing.

Of course, inputs and outputs play symmetric roles in switch scheduling. When consuming buffers in the downstream direction, as with backpressure protocols, the danger is for many inputs to simultaneously occupy buffers with cells going to the same output: output contention delays cell motion. Conversely, when reserving buffers in the upstream direction, like we do here, the danger is for many outputs to simultaneously reserve space for cells to come from the same input: input contention delays cell arrivals. This is analogous to "bad synchronization" of round-robin pointers in the initial, suboptimal iSLIP idea [19]. What limits input contention in our case is that buffer reservations constitute a *second pass* through the fabric, after requests have traversed once from inputs to the per-output scheduler. Thus, the only way for an input to receive an excessive number of reservations from multiple outputs is for other inputs *not* to have sent any requests to those outputs. Our recent paper [20] studied this issue in a single-stage fabric equipped with small output queues. There, we found that, when at any given time each scheduler may have reserved space for multiple inputs, the bad effects of "synchronization" are confined; on the order of ten cells per output port sufficed there. We observe a similar result in this paper, except that the number of segments per output is higher in the present paper, partly due to the buffered crossbar organization, which partitions each output queue's space into many smaller per-input spaces[4].

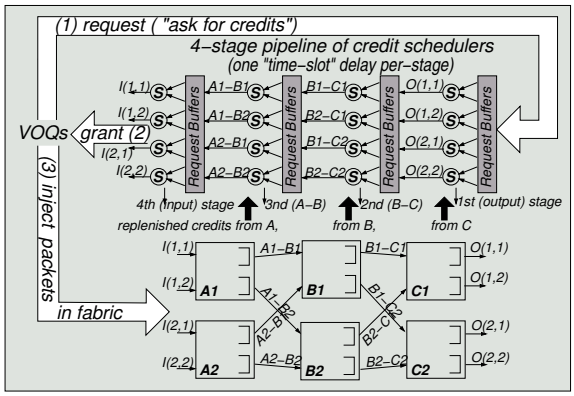

Fig. 2. pipelined buffer scheduling in a 4×4, three-stage non-blocking fabric.

Note that similar conflicting decisions also occur in buffered crossbar scheduling: a set of inputs may concurrently forward packets to a same output. However, these inputs are not blocked following their first, suboptimal decisions: they may continue sending cells to other outputs. This is the reason why buffered crossbars yield good performance without explicit coordination between the port schedulers. A buffered crossbar uses order N segment buffers per output to achieve this result. Our results indicate that one can do equally well with quite fewer (smaller) buffers.

3.2 Buffer Scheduling

Switch schedulers match inputs to outputs (or to internal links). Schedulers for bufferless switches do that precisely, per time-slot [19][13]. On the other hand, if there is a buffer of size B^o in front of each output (or internal) link, the scheduling constraint is relaxed: the amount of traffic admitted to that link can be as much as B^o per time-slot, but over any interval of length T that amount of traffic must not exceed $\lambda \cdot T + B^o$, where λ is the link rate[5].

We start with a conceptual scheduler, shown in figure 2, that admits this "window-type" feasible traffic; we will simplify it in the next sections. It consists of single-resource schedulers per output and per internal link. Each scheduler hands out credits for the buffer space in front of the corresponding link. Credits are replenished when the admitted cell eventually frees the corresponding resource. Each credit scheduler works independent of the others, using a private credit counter and private buffers (queues) that hold outstanding requests, until the scheduler can serve these requests. Each scheduler needs to grant at least one segment per segment-time (as long as it

[4]we further improve performance by using a backpressure mechanism that prevents persistent buffer reservations that conflict on inputs –see section 4.4.

[5]as mentioned already, when buffer space is reserved for every cell in every buffer, backpressure is not needed and cells are never dropped; in the absence of backpressure, each link always empties its buffer at peak rate λ. Notice that this also yields an upper bound for cell delay through the fabric: number of stages, times buffer size per stage, divided by link rate λ.

has credits), in order to keep its associated link busy. It can grant credits faster than that for a while, but when it runs out of credits the grant rate will be dictated by the credit replenishment rate, *i.e.* the actual traffic rate on the link.

As seen in figure 2, these schedulers form a 4-stage pipeline, with stages decoupled by the request buffers. Each stage contains N schedulers. The first-stage schedulers allocate space for the N output buffers of the C-stage switches (figure 1). We call them *credit* schedulers, because they hand out credits. The 2nd-stage schedulers do so for the B switches; the 3rd stage handles A-switch outputs; we call those *intermediate* schedulers. Finally, each 4th-stage scheduler corresponds to a linecard, and sends credits (grants) to the corresponding VOQs; we call them *grant* schedulers.

Credit schedulers enforce traffic admissibility (feasible rates). Due to multipath routing, credit (output) schedulers have the additional duty to perform path selection (choose a B switch), and direct the request to the appropriate 2nd-stage scheduler. When a grant is eventually sent to a linecard, it specifies both the output port (VOQ) and the route to be followed.

Let d^{cr}_{sch} denote the delay incurred by any single scheduler, and d^{pip}_{sch} the delay of a complete scheduling operation; $d^{pip}_{sch} = 4 \cdot d^{cr}_{sch}$. If each scheduler starts with an initial pool of at least $d^{pip}_{sch} \cdot \lambda$ worth of buffer-space credits, the pipeline can be kept busy, and throughput is not wasted. It suffices for schedulers to generate grants at rate λ. This is the nice feature of buffered fabrics: the control subsystem can be pipelined, with considerable inter-stage and total latency, as long as the pipeline rate (individual scheduler decision rate) matches link rate (one grant per segment-time).

3.3 Simplifications owing to Load Balancing

Route selection, for this multipath fabric, can be performed by the (per-output) credit schedulers. To obtain non-blocking operation, each (per input-output pair) flow must be distributed uniformly across all B switches. Such load balancing *(i)* has been shown very effective in Clos/Benes networks [14] [2], and *(ii)* can be implement in a distributed manner.

Consider a particular fabric-output port, o. Assuming an ideal, fluid distribution of the type discussed above, the traffic destined to output o and assigned to any particular switch B_b is $(\frac{\lambda}{M}, \frac{B^o}{M})$ leaky-bucket regulated. Now, considering all M outputs residing in the same C switch with output o, C_c, their collective traffic steered on any switch B_b will be the summation of M sources, each $(\frac{\lambda}{M}, \frac{B^o}{M})$ regulated, *i.e.* during any time interval T, the traffic admitted for any particular $B_b \rightarrow C_c$ link is:

$$L(B_b \rightarrow C_c, T) \leq \sum_{\nu=1}^{M} \frac{\lambda \cdot T + B^o}{M} = \lambda \cdot T + B^o$$

In other words, C switch admissions and load distribution guarantee that the aggregate traffic into the buffer in front of link $B_b \rightarrow C_c$ will always be (λ, B^o) constrained, independent of B switch admissions. At the same time, as already mentioned, there is no backpressure in the system of figure 2; hence link $B_b \rightarrow C_c$ will never be idle whenever its buffer is backlogged. Thus, in this ideal, fluid system, the traffic admitted into C switches will always find room in the $B_b \rightarrow C_c$ buffer, hence we can safely eliminate the second scheduler stage, which was responsible for securing buffers in the B switches. In a real (non-fluid) system, segment distribution will have quantization imbalance; thus, to prevent occasional overflows, we have to use backpressure from stage B to stage A.

To simplify the scheduler further, we discard the third scheduler stage (for A buffers) too, replacing it with conventional backpressure from stage A to the ingress linecards. We may safely do so because, in a fluid model, owing to perfect load balancing, the traffic entering the fabric and routed through any particular $A_a \rightarrow B_b$ link, is: $L(A_a \rightarrow B_b, T) \leq \sum_{\nu=1}^{M} \frac{\lambda}{M} = \lambda$. Although in the fluid model no A buffers (in front of $A_a \rightarrow B_b$ links) are needed, the real system does require them, in order to deal with quantization imbalance (multiple inputs of a same A switch sending concurrently to a same B switch, which is inevitable under distributed and independent load-balancing[6]).

These points are supported by the simulations results on delay under congestion epochs (sec. 6.3). The central scheduler described in the next section uses these simplifications: only credit (output) and grant (input) schedulers are needed, without any intermediate schedulers, as shown in figure 1. Note however that distributed scheduler implementations would need these intermediate nodes, in order for them to route grants from the credit schedulers to the grant schedulers (similar routing would be needed from VOQs to credit (output) schedulers).

4 . CENTRAL SCHEDULER

The scheduler proposed in section 3 is amenable to distributed implementations, scaling to large fabric valencies. However, in this paper, our reference design (section 1.2) employs a single *central* control chip, that contains all credit and grant schedulers. This choice allows "plain" switches in the datapath, without requiring modifications to add parts of the (distributed) scheduler in them[7]. This section shows *(i)* how to minimize the information carried by each request/grant notice, thus reducing control bandwidth, and *(ii)* how to turn each request and grant queue into a simple counter; it also presents the overall operation of the system.

4.1 Distribution Policy & Buffer Allocation Granularity

Section 3 used the term B^o to refer to the buffer in front of a switch output. Since we assume buffered crossbar switching elements, B^o is in fact partitioned per-input link of the switch; we will use the term B^x for an individual crosspoint buffer; the sizes are: B^o=$B^x \cdot M$. Since each C switch buffer corresponds to a specific upstream B switch, when a credit scheduler reserves space for a cell, it must choose a particular B switch

[6]Reference [21] removes these buffers by considering coordinated, static cell distribution from the input side, independent of the destination. However, this may cause Benes to be blocking.

[7]note, however, that it is also possible to implement distributed scheduling entirely on the linecards, without adding scheduler components in the switches.

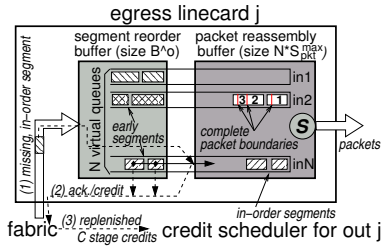

Fig. 3. Egress linecard organization. Segments are ordered, and then their fabric-output credits are returned to the credit scheduler.

and reserve space in the corresponding C buffer. Hence, each grant must carry a B-switch identifier.

4.1.1 Coordinated Load Distribution Decisions: We can perform this B switch choice using per-flow round-robin segment distribution[8]. Besides other advantages, this distribution method ensures that the route of each segment can be independently and consistently determined at both its (per-output) credit scheduler, and at its ingress linecard. Thus, this route assignment need not be communicated from the scheduler to the ingress: upon receiving a grant, the linecard can infer the route assigned to the segment by the credit scheduler. To do so, both of those units initialize a private, per-flow pointer to an agreed upon B switch, and then advance that pointer for every new grant or segment of that flow. In this way, we reduce the grant width by $\log_2 M$ bits.

4.1.2 Buffer Reservations: fixed or variable space?: To support variable-size segments, one has the option of either *(i)* having each request-grant transaction explicitly specify a size and carry the corresponding count; or *(ii)* always request and allocate buffer space for a maximum-size segment, even when the real segment that will eventually travel through that space is of a smaller size. We opt for fixed size allocation, for simplicity reasons: in this way, we reduce the width of requests and grants (they do not need to carry a size field), the width of request and credit counters in each scheduler, and the width of credits returned from C switches. But, most importantly, this method allows the grant queues in front of the (per-input) grant schedulers to be implemented as simple counters[9].

4.2 Resequencing: Bounding the Reorder Buffer Size

Multipath routing through the B switches can deliver packets out of order to the C switches. Resequencing is performed on the egress linecards[10], as shown in figure 3. The scheduler bounds the required reorder buffer size, and that bound is very modest in the reference design, as discussed in section 1.4.

[8]this method ignores the (usually small) load imbalance caused by variable segment size.

[9]given the round-robin way in which ingress linecards infer each segment's route, grant schedulers are not allowed to merge consecutive grants. If grants were variable-size, a simple counter would not suffice to keep consecutive grants from being merged with each other.

[10]resequencing could also be performed in the C switches, but *not* with the existing buffers: to avoid deadlock, the reorder buffers must be in addition to the already existing (single-lane, crosspoint) buffers.

We let the credit schedulers manage and allocate space in the reorder buffers, just as they do for the buffers in the C switches. The C buffers total size is B^o per output port. We assume that each egress linecard has a reorder buffer of equal size, $B^o = B^x \cdot M$. In this way, by allocating space for a segment in the C switch, the credit scheduler also implicitly allocates space in the reorder buffer. Next, we modify the time at which a credit is returned to the central scheduler: up to now we assumed a credit is generated as soon as a segment exits the fabric; in reality, the credit is only generated when the segment is *no longer waiting* for any earlier segment to arrive from the fabric. This scheme effectively combines flow-control and resequencing, using a common admission mechanism in the central scheduler. Since we delay credit generation, the added delay (C switch to end of resequencing) must be counted in the overall control round-trip time (RTT), to be used in sizing fabric buffers. The next section reviews overall system operation.

4.3 Operation Overview

Segment Admission: In each ingress linecard, a *request scheduler* visits the VOQs and sends requests for the corresponding outputs to the central scheduler. Upon reaching the latter, each request, say $i \rightarrow o$, increments the $i \rightarrow o$ request count, which is maintained in front of the *credit scheduler* for output o. The credit scheduler also maintains M credit counters, one per crosspoint queue in its C switch, and N distribution pointers, one per flow arriving to this output. Each credit counter is decremented by one when the credit scheduler allocates space from that counter to one cell (segment). Each distribution pointer identifies the B switch through which to route the next cell of the corresponding flow, $i \rightarrow o$; it is initialized and incremented as described in section 4.1.1. Connection $i \rightarrow o$ is eligible for service at its (output) credit scheduler, when its request counter is non-zero, and the credit counter pointed by the distribution counter $i \rightarrow o$ is also non-zero. Once connection $i \rightarrow o$ gets served, its request count decreases by one, and a grant is routed to its input *grant scheduler*, where it increments grant counter $i \rightarrow o$. Any non-zero grant counter is always eligible for service, and, once served, is decremented by one. When served, grant $i \rightarrow o$ is sent to its ingress linecard, to admit a new segment inside the fabric.

Segment Injection: When a grant arrives to a VOQ, that queue injects its head segment into the fabric. One segment is injected even if its size is not the maximum (grants always refer to maximum-size segments). Small segments underutilize the buffer spaces that have been reserved for them; this is not a problem: the reason why VOQ $i \rightarrow o$ does not contain a maximum-size segment is that this smaller segment is the only datum in the queue [11]. If the load of this flow persists, the VOQ will grow to contain multiple packets, in which case the head segment will always be of maximum-size.

The route of the injected segment is given by input distribution counter $i \rightarrow o$; this counter is initialized and incremented as described in section 4.1.1. A sequence tag is included in the

header of the segment, specifying its order among segments in the $i \rightarrow o$ VOQ. The segment has then to compete against other "granted" segments, and will reach its C switch subject to hop-by-hop, credit-based backpressure[11]. This backpressure is indiscriminate (not per-flow), but, as explained in section 3.3, it will not introduce harmful blocking. No backpressure is exerted from the egress linecards to the C stage.

4.4 Limiting the per-flow Outstanding Requests

The request schedulers limit the number of requests that a VOQ may have outstanding inside the central scheduler to an upper bound u. This has the following benefits. First, the respective request or grant counter in the central scheduler will never wraparound (overflow) if it is at least $\lceil log_2 u \rceil$-bit wide. Second, this "flow control" prevents output credit schedulers from severely synchronizing in conflicting decisions, *i.e.* granting buffers to a few, oversubscribed inputs. If some credit schedulers do so for a while, their grants will wait in front of the input grant schedulers; the latter hand them out at the rate of one per segment time; the busy VOQ cannot generate new requests until it receives grants. Effectively, after a while, the synchronized output (credit) schedulers will find that the "congested" inputs have no more requests for them; therefore they will be forced to serve requests from other inputs.

5 . CENTRAL SCHEDULER IMPLEMENTATION

5.1 Central Chip Bandwidth

As shown in figure 1, requests issued by the ingress linecards first travel to the A-switch, at the rate of one VOQ request per-segment-time; each request carries an identifier of the output port it refers to. Inside the A switch, VOQ requests are time division multiplexed (TDM) upon a link that transfers M (equal to TDM frame size) requests to the scheduler per-segment-time, one from each linecard –see figure 4. The scheduler infers the input port linecard of a request by its position in the TDM frame (request belt). Grants destined to input ports of a particular A switch depart from the scheduler on a similar TDM link (grant belt): the position (slot) of a grant in a TDM frame indicates to the A-switch the linecard that must receive each grant. The "payload" of each request or grant notice is an identifier of the fabric-output port that the notice goes to, or comes from; this destination identifier can be encoded in $log_2 N$ bits. Besides request-grant notices, the central scheduler must also receive credits from the switches in the C stage. These credits are conveyed through a link connecting each C-switch with the scheduler. Each such link carries M credits per-segment-time, one per output port, in a similar TDM manner: the position of a credit in the TDM frame identifies the output port that the credit comes from, and its payload specifies the crosspoint queue ahead of that port generating the credit –*i.e.*, $\log_2 M$ bits payload per TDM slot.

[11] our system uses credit-based backpressure from the C stage to the B stage, so that a few segments can be injected without requesting credits –see section 1.5. Assuming that all injected segments have been granted credits, this backpressure will never block or halt a segment.

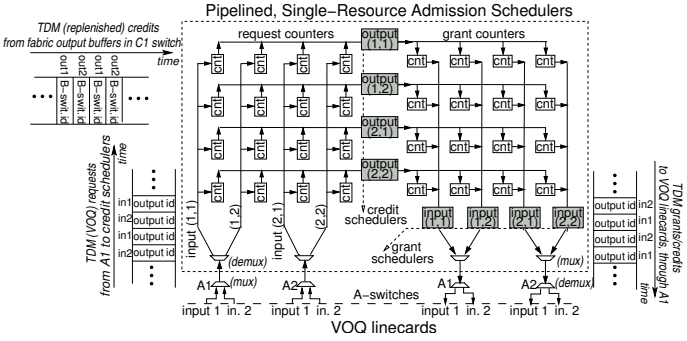

Fig. 4. Central scheduler for N=4 and M=2; request-grant communication with ingress linecards and credit replenishment from C switches use time division multiplexing (TDM).

Using TDM multiplexing, the aggregate bandwidth of the scheduler's chip is $2 \cdot N \cdot \log_2 N$ bits per-segment-time –for receiving requests from and issuing grants to all fabric-input ports– plus $N \cdot \log_2 M$ bits per-segment-time for receiving credits from all fabric-outputs, for a total of $N \cdot (2 \cdot \log_2 N + \log_2 M)$ bits per-segment-time. For a 1024-port fabric, with M=32, λ=10 Gb/s, and segment size 64 Bytes, the aggregate input plus output bandwidth is 25.6 Kbit per 51.2 ns, or roughly equal to 500 Gb/s.

5.2 Routing Requests and Grants to their Queues

Figure 4 depicts the internal organization of the central scheduler. Requests from different ingress ports arriving through a given A-switch are conceptually demultiplexed and routed to their (output) credit scheduler; in an actual implementation, no demultiplexing is needed, and the counter can be held in SRAM blocks, accessed in the same TDM manner as external links. Conceptually, at the interface of the scheduler's chip, we have N inputs that want to "talk" to N output schedulers. As shown in figure 4, this can be implemented by a crossbar. Each crossbar input needs only identify the output for which it has a new requests –no other payload is being exchanged. When a credit (output) scheduler (middle of the chip) serves an input it increments the corresponding grant counter (right half of the chip). The grant counters form another conceptual "crossbar" analogous to the one formed by the request counters.

This first, conceptual organization uses $2 \cdot N^2$ request/grant counters, $\lceil log_2 u \rceil$-bit wide each, and N^2, $log_2 M$-bit wide distribution (load balancing) counters. For a 1024-port fabric, and u=32, this results in roughly 15 M of counter bits. Each such bit costs about 20 transistors *(xtors)*, hence, the chip of such a straightforward implementation would need approximately 300 M*xtors*, in total.

A better implementation groups several counters in sets, implemented as SRAM blocks with external adders for increments and decrements. In this way, we can reduce the chip die considerably, since SRAM is much more compact than random logic. At the same time, input or output operations are time multiplexed on a set of hardware controllers running faster than the segment-time. For instance, we can group outputs in

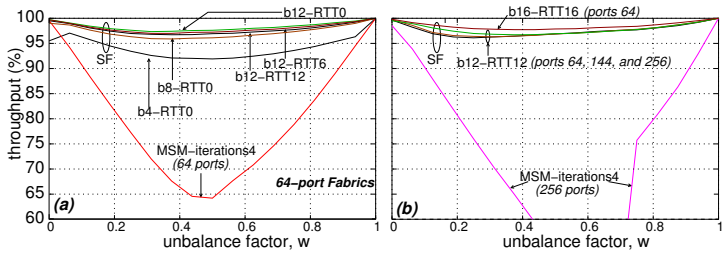

Fig. 5. Throughput under unbalanced traffic. 100% input load consisting of fixed-size cell Bernoulli arrivals. *(a)* 64-port fabric (N=64, M=8); *(b)* varying fabric sizes up to N=256, M=16.

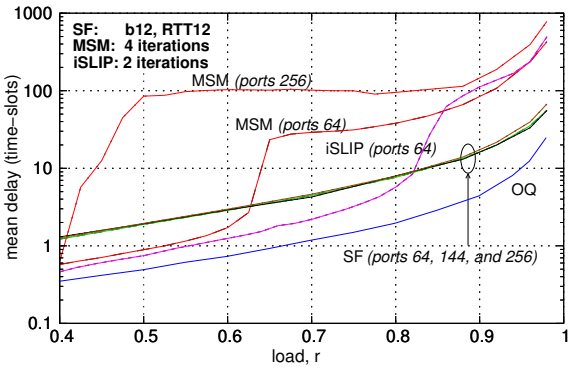

Fig. 6. Delay versus input load, for varying fabric sizes, N; buffer size b=12 cells, RTT=12 cell-times (time-slots). Uniform Bernoulli fixed-size cell arrivals. Only the queueing delay is shown.

groups of M, and use only one credit scheduler controller performing request admissions for these outputs, in a TDM manner.

6 . Simulation Results

An event-driven simulation model was developed in order to verify the design and evaluate its performance for various fabric sizes, crosspoint buffer sizes, and round-trip times. All experiments except section 6.6 use fixed-size cell traffic, in order to compare to alternative architectures that only work with fixed-size cells. We simulated the fabric under smooth, bursty, unbalanced, hotspot, and inadmissible traffic. Smooth traffic consists of Bernoulli cell arrivals with uniformly distributed destinations. Bursty traffic is based on a two-state (ON/OFF) Markov chain[12]. In unbalanced traffic, destinations are picked as in [22], using a parameter w: when w is zero, traffic is uniform, whereas when w is one, traffic consists of persistent, non-conflicting, input-output connections. Under hotspot traffic, each destination belonging to a designated set of "hot spots" receives traffic at 100% collective load, uniformly from all sources; the rest of the destinations receive a smaller load. Hotspots are randomly selected among the fabric-output ports. Finally, inadmissible traffic patterns were used in order to examine how well can our architecture distribute input and output port bandwidth based on sophisticated QoS criteria.

The delay reported is the average, over all segments (cells), of the segment's exit time –after being correctly ordered inside the egress resequencing buffers–, minus the segment's birth time, minus the request-grant cold start delay, and minus the segment's sojourn time through the fabric. Thus, under zero contention, the reported delay of a segment can be as small as zero. The control round-trip time (RTT) (figure 1: arrows 2,3,4), consists of: credit and grant scheduler delays plus segment sojourn time from ingress to egress plus credit sojourn time from egress linecard to scheduler. This control RTT is used to size the crosspoint buffers. We use 95% confidence intervals of 10% in delay experiments, and of 1% in throughput experiments.

[12]ON periods (consecutive, back-to-back cells arriving at an input for a given output) last for at least one (1) cell-time, whereas OFF periods may last zero (0) cell-times, in order to achieve 100% loading of the switch. The state probabilities are calibrated so as to achieve the desirable load, giving exponentially distributed burst length around the average indicated in any particular experiment.

We use the name *Scheduled Fabric (SF)* to denote our system. *SF* uses no internal speedup. The default schedulers in *SF* are pointer-based round-robin (RR) schedulers throughout the fabric, the linecards, and the admission unit[13]. We compare *SF* to output queueing (OQ), to *i*SLIP, and to a three-stage Clos fabric consisting of a bufferless middle stage, and buffered first and last stages (*MSM*), scheduled using the *CRRD* algorithm [13].

6.1 Throughput: Comparisons with MSM

First, we measure throughput for different crosspoint buffer sizes, b, and for different RTTs under unbalanced traffic; for comparison, we also plot *MSM* results. See figure 5. Figure 5(a) shows that with b as small as 12 cells, *SF* approaches 100% throughput under uniform traffic (w=0), and provides more than 95% throughput for intermediate w values, which correspond to unbalanced loads. We also see that, with the same buffer size (b=12), and for any RTT up to 12 segment-times, this performance does not change. Figure 5(b) shows this performance to stay virtually unaffected by the fabric size, N, increasing from 64 to 256. By contrast, the performance of *MSM* drops sharply with increasing N. Although *MSM* may deliver 100% throughput (similar to *i*SLIP), it is designed to do that for the uniform case, when all VOQs are persistent; if some VOQs fluctuate however, pointers can get synchronized, thus directly wasting output slots. By contrast, *SF* does not fully eliminate packet conflicts; in this way, every injected segment, even if conflicting, makes a step "closer" to its output, thus being able to occupy it on the first occasion.

6.2 Smooth Arrivals: Comparison with OQ, iSLIP, and MSM

Figure 6 shows the delay-throughput performance of *SF* under smooth traffic, and compares it with that of *MSM*, *i*SLIP, and ideal OQ switch. Compared to the bufferless architectures, *SF* delivers much better performance. The delay of *SF* is not affected by fabric size, while that of *MSM* is very much affected. The delay of *SF* under smooth traffic is within four

[13]the round-robin pointer of each credit scheduler visits inputs (request counters) in a pseudo-random but preprogrammed order, different for each output, for reasons of better "desynchronization".

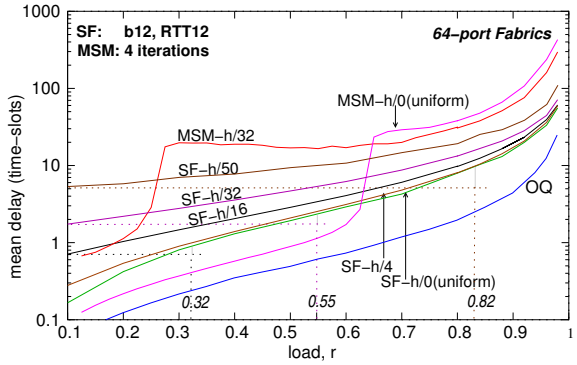

Fig. 7. Delay of well-behaved flows in the presence of hotspots. h/• specifies the number of hotspots, *e.g.*, h/4 corresponds to four hotspots. Bernoulli fixed-size cell arrivals; 64-port fabric, b=12 cells, RTT=12 cell-times (time-slots). Only the queueing delay is shown.

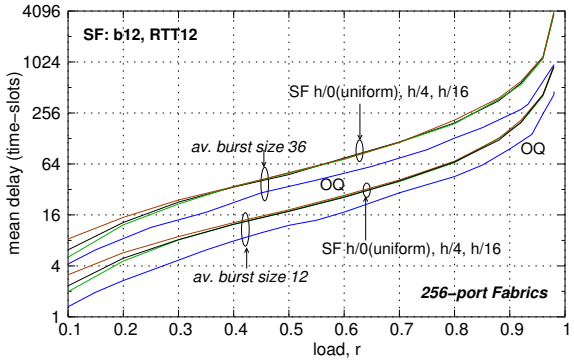

Fig. 8. Delay of well-behaved flows in the presence of hotspots, for varying burstiness factors. Bursty fixed-size cell arrivals; 256-port fabric, b=12 cells, RTT=12 cell-times (time-slots). Only the queueing delay is shown.

times that of OQ. We hypothesize that the main source of additional delay, in *SF* relative to OQ, is the large number of contention points that a segment goes through during its trip inside the fabric.

6.3 Overloaded Outputs & Bursty Traffic

A major concern in multistage fabrics is the adverse effect that congestion at certain outputs may have on other uncongested outputs. Our design explicitly guards against that danger. Figure 7 presents the delay of uncongested flows (non-hotspot traffic), in the presence of a varying number of other congested outputs (hotspots). All flows, congested or not, are fed by Bernoulli sources. For comparison, we also plot cell delay when no hotspot is present, denoted by h/0, and the OQ delay.

To see how well *SF* isolates flows, observe that the delay of h/4 (*i.e.*, the delay of well-behaved flows in the presence of four (4) congested outputs) is virtually *identical* to that of h/0. Nevertheless, we see the delay of well-behaved flows increasing with the number of hotspots, with the increase being more pronounced for large numbers of hotspots. If the well-behaved flows were subject to backpressure signals coming from queues that feed oversubscribed outputs, these flows' delay could probably grow without bound, even at very moderate loads. However, this is not the case with *SF*. The observed increase in delay is *not* due to congestion effects, but to hotspot traffic increasing the contention along the shared paths inside the fabric. For instance, when fifty out of the sixty-four output ports of the fabric are oversubscribed (h/50), and the load of the remaining fourteen output flows is 0.1, the effective load, at which each fabric-input injects segments, is close to 0.82. We have marked in figure 7 the delays of h/50 at load 0.1 and of h/0 at load 0.82. We see that these two delays are almost identical[14]! Analogous behavior can be seen in the *MSM* plots. (Consider that *MSM* contains N large VOQs inside each A-switch, which are being shared among all upstream ingress linecards, in order to isolate output flows.)

Not shown in the figure is the utilization of the hotspot destinations (the load offered to them is 100%). In *SF*, all hotspots were measured to be 100% utilized, for any load of the well-behaved flows; by contrast, in *MSM*, the respective utilization dropped below 100%, because, for 100% utilization, the prerequisite of the *CRRD* scheme is to desynchronize its RR pointers, which can only be achieved when all VOQs are active. When some VOQs are not always in active state, as those belonging to the well-behaved flows in our experiment here, pointers may get synchronized, rendering considerable throughput losses.

Lastly, we examine the effect of burstiness on the performance of the *SF* fabric[15]. The results[16] are shown in figure 8. While *SF* delay was approximately 4 times larger than OQ delay under smooth traffic, here *SF* delay is only 1.5 times larger than OQ delay[17]. In most non-blocking fabrics, the primary source of delay under bursty traffic is the severe (temporal) contention for the destination ports, many of which may receive parallel bursts from multiple inputs [23][19]. For the same reason, under bursty traffic, the incremental delay that well-behaved flows experience in the presence of hotspots is less pronounced than with Bernoulli arrivals –figure 8 versus figure 7.

6.4 Output Port Bandwidth Reservation

The *SF* architecture not only protects one output flow from another, but can also differentiate among flows going to the same output, if so desired for QoS purposes. Previous experiments used RR schedulers. Now, we modify the (single-resource) credit schedulers, which allocate output-port bandwidth to competing inputs: in this experiment we use

[14]the delay of h/50 at load 0.1 is actually a bit lower than the delay of h/0 at load 0.82. This is so because in h/50 under (non-hotspot) load 0.1, the *output* load for the uncongested packets, whose delays we measure, is 0.1, whereas in h/0 under load 0.82, the output load is 0.82 for all flows.

[15]in [20], using a similar scheduler but for single-stage switches, larger delays than those that we report here were observed under bursty traffic; [20] eliminated these large delays using a method that, if needed, can easily be incorporated in the scheduler of this paper, as well.

[16]we used a warm-up period of 150 millions cells before gathering delay samples.

[17]the same performance trends can be found in [2].

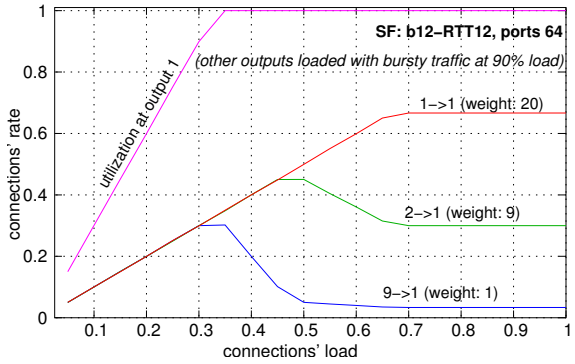

Fig. 9. Sophisticated output bandwidth allocation, using WRR/WFQ credit schedulers; 64-port fabric, b=12 segments, RTT=12 segment-times.

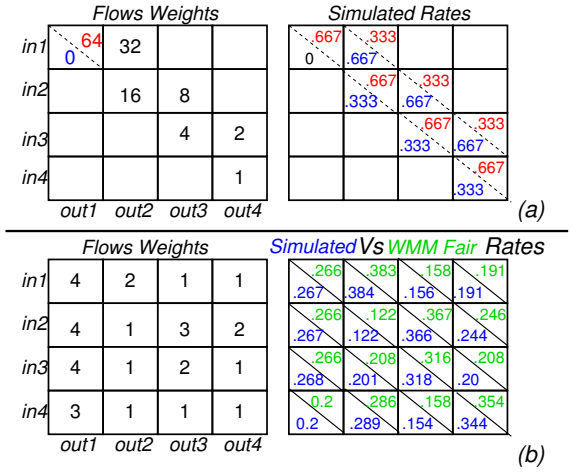

Fig. 10. Weighted max-min fair allocation of input and output port bandwidth, using WRR/WFQ schedulers; 4-port fabric, b=12 segments, RTT=12 segment-times.

WRR/WFQ credit schedulers[18].

In figure 9, we configured three flows (connections) in a 64-port fabric, flows 1→1, 2→1, and 9→1, with weights of twenty (20), nine (9), and one (1), respectively; each of them is the only flow active at its input, and the load it receives changes along the horizontal axis. Inputs other than 1, 2, and 3, receive a uniform, bursty (background) traffic, at 0.9 load, targeting outputs 2 to 64. The vertical axis depicts connections' normalized service (rate), measured as cell rate at the output ports of the fabric.

As figure 9 shows, when the demand for output 1 is feasible (up to 0.33 load per flow), all flows demands are satisfied. At the other end, when all flows are saturated (starting from 0.66 load per flow), each flow gets served at a rate equal to its fair share –i.e., 0.66, 0.30, and 0.033. When the load of a flow is below this fair share, the bandwidth that stays unused by this flow gets distributed to the other flows, in proportion to those other flows' weights.

6.5 Weighted Max-Min Fair Schedules

In this section, we place WRR (output) credit schedulers, as we did in section 6.4, and WRR (input) request schedulers. Each VOQ flow $i \rightarrow j$ has a unique weight; this weight is being used by the WRR request scheduler at ingress i, as well as by the WRR credit scheduler for output port j. We model persistent VOQ –either (active) constantly full or constantly empty– flows, and we measure their rate. Each active VOQ is fed with back-to-back cells arriving at the line rate. (u is set equal to 32, thus the request rate of a VOQ connection gets equalized to its grant (service) rate by bounding the per-VOQ number of pending requests –see section 4.4.)

First, we configure a "chain" of dependent flows as in [24]. The left table in figure 10(a) depicts the weights of the flows in a 4×4 fabric comprised of 2×2 switches. When flow 1→1 is active, with a weight of 64, its weighted max-min fair (WMMF) share is 2/3, and each subsequent connection along the diagonal of the table deserves a WMMF rate of 1/3, 2/3, 1/3, etc[19]. Service rates are shown in the table on the right (upper corner in each box): the rates that the *SF* fabric assigns to connections exactly match their WMMF shares. When 1→1 is inactive, with zero weight, the fair shares of the remaining connections get reversed, becoming 2/3, 1/3, 2/3, etc. As shown in the bottom corner of each box in the table, again simulation rates exactly match these new WMM fair shares.

In figure 10(b) we configured 16 active connections in the 4-port fabric; their weights, their WMMF shares, as well as the *SF* simulated rates are shown in the tables. We again see how close the rate allocation of the *SF* fabric approximates the ideal WMMF allocation.

6.6 Variable-Size Multipacket Segments

Our results up to now assumed fixed-size cell traffic. In this last experiment, we present simulations of variable-size multi-packet segments that carry the payload of variable-size packets. These experiment use reassembly buffers inside the egress linecards, to form complete packets from segments, after the latter depart from the reorder buffer. We assume 10 Gbps sources sending variable-size packets uniformly over all destinations, with exponential inter-arrival times (Poisson packet sources). Packet sizes follow the Pareto distribution, ranging from 40 up to 1500 Bytes. Maximum segment is 260 Bytes, and minimum is 40. We compare the *SF* architecture to a buffered crossbar with 2 KBytes per crosspoint buffer and no segmentation or reassembly similar to the architecture [12].

Our results are shown in figure 11. We see that *SF* delivers comparable delays to these of the buffered crossbar; at low loads, the dominant delay factors in *SF* are the VOQ delay, – in this experiment, VOQ delay includes 500 ns of request-grant cold-start delay–, as well as the reordering and the reassembly delays. Excluding the request-grant delay overhead, the delay of *SF* is within 2 to 3 times of the delay of "ideal" buffered crossbars, that directly operate on variable-size packets.

[18] all other schedulers within the fabric are left intact (RR).

[19] for algorithms computing WMM fair schedules see [25].

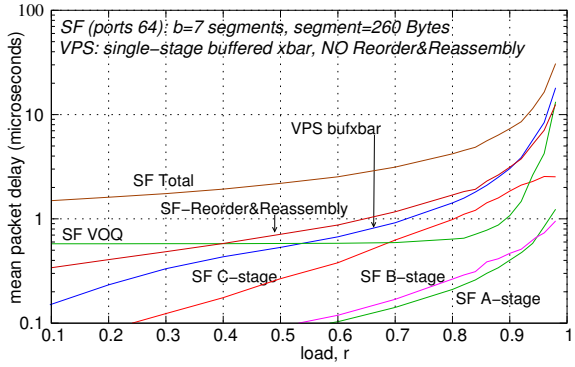

Fig. 11. Packet delay performance under variable-size packet arrivals, using variable-size multipacket segments; uniform Poisson packet arrivals on 10 Gbps input lines; b=7 (maximum-size) segments. Packet delay includes the request-grant delay, segment scheduling and propagation delays, as well as segment reordering and packet reassembly delays.

7 . Conclusions & Future Work

We proposed and evaluated a novel, effective, and realistic scheduling architecture for non-blocking buffered switching fabrics. It relies on multiple, independent, single-resource schedulers, operating in a pipeline. It unifies the ideas of central and distributed scheduling, and it demonstrates the latency - buffer space tradeoff. It provides excellent performance at realistic cost.

Work in progress includes completing our study of variable-size segment performance, designing schemes to route back-pressure and control signals more efficiently, and simulating various protocols that eliminate the request-grant delay under light load. Other future work includes a careful comparison against the RECN protocol, a study of other scheduling disciplines besides RR or WFQ/WRR (as that proposed in [26] for networks with finite buffers), and a study of multicast capabilities.

8 . Acknowledgments

This work was supported by an IBM Ph.D. Fellowship. The authors would especially like to thank Georgios Sapountzis for passing on his understanding of Benes networks, as well as Kostas Harteros and Dionisis Pnevmatikatos for their valuable contribution in implementation issues. The authors also thank, among others, Paraskevi Fragopoulou, Georgios Georgakopoulos, Ioannis Papaefstathiou, Vassilios Siris, and Georgios Passas for helpful discussions.

References

[1] G. Kornaros, e.a.: "ATLAS I: Implementing a Single-Chip ATM Switch with Backpressure", *IEEE Micro*, vol. 19, no. 1, Jan/Feb. 1999, pp. 30-41.

[2] G. Sapountzis, M. Katevenis: "Benes Switching Fabrics with O(N)-Complexity Internal Backpressure", *IEEE Communications Magazine*, vol. 43, no. 1, January 2005, pp. 88-94.

[3] M. Katevenis, D. Serpanos, E. Spyridakis: "Credit-Flow-Controlled ATM for MP Interconnection: the ATLAS I Single-Chip ATM Switch", *Proc. 4th IEEE Int. Symp. High-Perf. Computer Arch. (HPCA-4)*, Las Vegas, NV USA, Feb. 1998, pp. 47-56; http://archvlsi.ics.forth.gr/atlasI/

[4] J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia, T. Nachiondo: "A New Scalable and Cost-Effective Congestion Management Strategy for Lossless Multistage Interconnection Networks", *Proc. 11th IEEE Int. Symp. High-Perf. Computer Arch. (HPCA-11)*, San Francisco, CA USA, Feb. 2005, pp. 108-119.

[5] D. Stephens, Hui Zhang: "Implementing Distributed Packet Fair Queueing in a Scalable Switch Architecture", *IEEE INFOCOM'98 Conference*, San Francisco, CA, Mar. 1998, pp. 282-290.

[6] N. Chrysos, M. Katevenis: "Weighted Fairness in Buffered Crossbar Scheduling", *Proc. IEEE HPSR'03*, Torino, Italy, pp. 17-22; http://archvlsi.ics.forth.gr/bufxbar/

[7] W. Kabacinski, C-T. Lea, G. Xue - Guest Editors: 50th Anniversary of Clos networks –a collection of 5 papers, *IEEE Communications Magazine*, vol. 41, no. 10, Oct. 2003, pp. 26-63.

[8] V. Benes: "Optimal Rearrangeable Multistage Connecting Networks", *Bell Systems Technical Journal*, vol. 43, no. 7, pp. 1641-1656, July 1964.

[9] L. Valiant, G. Brebner: "Universal Schemes for Parallel Communication" *Proc. 13th ACM Symp. on Theory of Computing (STOC)*, Milwaukee, WI USA, May 1981, pp. 263-277.

[10] F. Chiussi, D. Khotimsky, S. Krishnan: "Generalized Inverse Multiplexing for Switched ATM Connections" *Proc. IEEE GLOBECOM Conference*, Australia, Nov. 1998, pp. 3134-3140.

[11] M. Katevenis, G. Passas: "Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures", *Proc. IEEE Int. Conf. on Communications (ICC 2005)*, Seoul, Korea, 16-20 May 2005, paper ID "09GC08-4"; http://archvlsi.ics.forth.gr/bufxbar/

[12] Manolis Katevenis, Giorgos Passas, Dimitris Simos, Ioannis Papaefstathiou, Nikos Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE ICC'04*, Paris, France, vol. 2, pp. 1090-1096; http://archvlsi.ics.forth.gr/bufxbar

[13] Eiji Oki, Zhigang Jing, Roberto Rojas-Cessa, H.J. Chao: "Concurrent Round-Robin-Based Dispatching Schemes for Clos-Network Switches", *IEEE/ACM Trans. on Networking* vol. 10, no. 2 December 2002.

[14] S. Iyer, N. McKeown: "Analysis of the parallel packet switch architecture", *IEEE/ACM Trans. on Networking*, 2003, 314-324.

[15] D. Khotimsky, S. Krishnan: "Stability analysis of a parallel packet switch with bufferless input demultiplexors", *Proc. IEEE ICC*, 2001 100-111.

[16] F. M. Chiussi, J. G. Kneuer, and V. P. Kumar: "The ATLANTA architecture and chipset", *IEEE Commun. Mag.*, December 1997, pp. 44-53.

[17] Prashanth Pappu, Jyoti Parwatikar, Jonathan Turner and Ken Wong: "Distributed Queueing in Scalable High Performance Routers", *Proc. IEEE Infocom*, March 2003.

[18] Prashanth Pappu, Jonathan Turner and Ken Wong: "Work-Conserving Distributed Schedulers for Terabit Routers", *Proc. of SIGCOMM*, September 2004.

[19] Nick McKeown: "The *i*SLIP Scheduling Algorithm for Input-Queued Switches" *IEEE/ACM Trans. on Networking*, vol. 7, no. 2, April 1999.

[20] N. Chrysos, Manolis Katevenis: "Scheduling in Switches with Small Internal Buffers", *Proc. IEEE Globecom'05*, St. Louis, MO, USA, 28 Nov. - 2 Dec. 2005; http://archvlsi.ics.forth.gr/bpbenes

[21] Xin Li, Zhen Zhou, and Mounir Hamdi: "Space-Memory-Memory Architecture for Clos-Network Packet Switches", *Proc. IEEE ICC'05*, Seoul, Korea, 6 pages.

[22] R. Rojas-Cessa, E. Oki, H. Jonathan Chao: "CIXOB-k: Combined Input-Crosspoint-Output Buffered Switch", *Proc. IEEE GLOBECOM'01*, vol. 4, pp. 2654-2660.

[23] S. Q. Li: "Performance of a Nonblocking Space-Division Packet Switch with Correlated Input Traffic", *IEEE Trans. on Communications* , vol. 40, no. 1, Jan. 1992, pp. 97-107.

[24] N. Chrysos, M. Katevenis: "Transient Behavior of a Buffered Crossbar Converging to Weighted Max-Min Fairness", *Inst. of Computer Science, FORTH*, August 2002, 13 pages; http://archvlsi.ics.forth.gr/bufxbar/

[25] E. Hahne: "Round-Robin Scheduling for Max-Min Fairness in Data Networks", *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 7, September 1991.

[26] Paolo Giaccone, Emilio Leonardi, Devavrat Shah: "On the Maximal Throughput of Networks with Finite Buffers and its Application to Buffered Crossbars", *IEEE INFOCOM Conf.*, Miami USA, March 2005.

