The Switch Reordering Contagion: Preventing a Few Late Packets from Ruining the Whole Party

Ori Rottenstreich, Pu Li, Inbal Horev, Isaac Keslassy and Shivkumar Kalyanaraman

***Abstract*—Packet reordering has now become one of the most significant bottlenecks in next-generation switch designs. A switch practically experiences a *reordering delay contagion*, such that a few late packets may affect a disproportionate number of other packets. This contagion can have two possible forms. First, since switch designers tend to keep the *switch flow* order, i.e. the order of packets arriving at the same switch input and departing from the same switch output, a packet may be delayed *due to packets of other flows* with little or no reason. Further, *within a flow*, if a single packet is delayed for a long time, then all the other packets of the same flow will have to wait for it and suffer as well.**

In this paper, we suggest solutions against this reordering contagion. We first suggest several hash-based counter schemes that prevent inter-flow blocking and reduce reordering delay. We further suggest schemes based on network coding to protect against rare events with high queueing delay within a flow. Last, we demonstrate using both analysis and simulations that the use of these solutions can indeed reduce the resequencing delay. For instance, resequencing delays are reduced by up to an order of magnitude using real-life traces and a real hashing function.

***Index Terms*—Switching theory, Packet-switching networks.**

I. Introduction

A. Switch Reordering

Packet reordering is emerging as a significant design issue in next-generation high-end switch designs. While it was easier to prevent in older switch designs, which were more centralized, the increasingly distributed nature of switch designs and the growth in port numbers are making it harder to tackle [1].

Figure 1 illustrates a simplified $N \times N$ multi-stage switch architecture, with N input ports, M middle elements, and N output ports. It could for instance be implemented using either a Clos network or a load-balanced switch [2]. Variable-size packets arriving to the switch are segmented into fixed-size cells. Each such cell is load-balanced to one of the M middle elements, either round-robin or uniformly-at-random. It is later sent to its appropriate output. There, in the *output resequencing buffer*, it waits for late packets from other middle elements, so as to depart the switch in order.

For instance, assume that cells A through F arrive in order to the switch, and belong to the same switch flow, i.e. share the same input and output. When A arrives to the output, it can depart. However, if cells B and C still wait to be transmitted out of their middle elements, then cells D through F cannot depart the switch, because they wait for B and C to be sent in order.

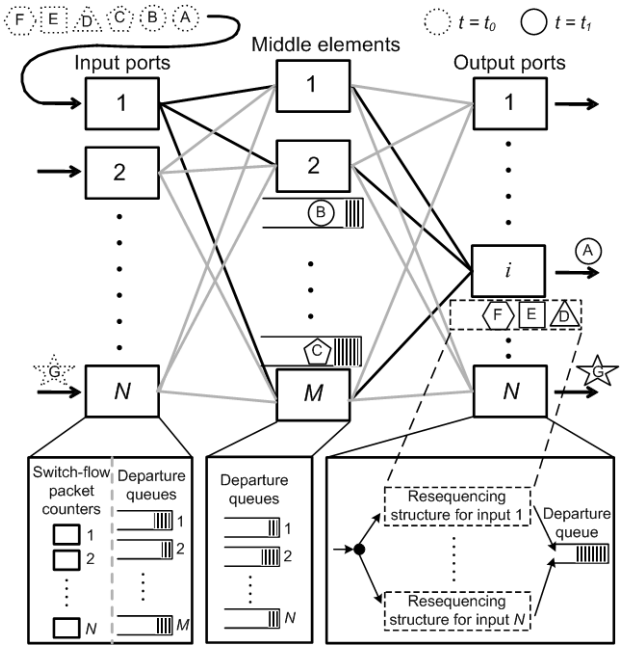


Fig. 1. Switch flow blocking in the switch architecture.

More generally, packet reordering occurs when there is a difference in delay between middle elements. In practice, this can happen because of many reasons: e.g., a temporary congestion at a middle element following the sudden arrival of a few high-fanout multicast packets from different inputs, a periodic maintenance algorithm, or a flow-control backpressure message that temporarily stops transmissions out of a middle element but has not yet stopped arrivals to it [3].

Because of packet reordering, a large delay at a single middle element can affect the whole switch. For instance, assume that a given middle element temporarily stops transmitting cells because of a flow-control message. Consider an arbitrary switch flow, i.e. a set of packets sharing the same switch input and output. If the switch-flow cells are load-balanced in round-robin order across middle elements, and the switch flow has at least M cells, then at least one cell will be stuck at the middle element. In turn, this will block the *whole* switch flow, since all other subsequent cells are waiting for it at the output resequencing buffer. Therefore, the switch will behave as if *all* the middle elements were blocked. Instead of affecting only $\frac{1}{M}^{th}$ of the traffic, it potentially affects the whole traffic.

Reordering causes many issues in switch designs. The

O. Rottenstreich, P. Li, I. Horev and I. Keslassy are with the Department of Electrical Engineering, Technion, Haifa 32000, Israel (e-mails: {or@tx, puli@tx, ihorev@tx, isaac@ee}.technion.ac.il).

S. Kalyanaraman is with IBM Research, India (e-mail: shivkumar-k@in.ibm.com).

cells that are waiting at the output resequencing buffer are causing both *increased delays* and *increased buffering needs*. In fact, in current switch designs, the resequencing buffer sizes are typically pushing the limits of available on-chip memory. In addition, reordering buffers cause *implementation complexity* issues. This is because the output resequencing structures, shown in Figure 1, are conceptually implemented as linked lists. Therefore, longer resequencing buffers imply that designers need to insert cells into increasingly longer linked lists at a small constant average time.

There are two ways to address these reordering issues. First, it is possible to change routing. One commonly-proposed solution to fight reordering is to *hash each flow to a random middle switch* [4]–[6]. However, while this indeed prevents reordering, it also achieves a significantly *lower throughput*, especially when dealing with bigger flows. In addition, other solutions rely on a careful synchronization of packets to avoid reordering [2], [4]. Yet these solutions are often too complex for typical high-performance routers with high internal round-trip-times and large packet rates. In this paper, we assume that *routing is given*, and do not try to improve it in any way.

A second approach, which is often adopted by switch vendors, is to use *significant speedups* on the switch fabrics (typically 2X, but published numbers go up to 5X [6], [7]). However, this approach also wastes a large part of the bandwidth. *Our goal is to suggest ways of scaling routers in the future without further increasing this speedup.*

B. Reordering Contagion and Flow Blocking

In this paper, we divide reordering effects into two different types, and suggest algorithms to deal with each type. More precisely, we consider two types of *reordering contagion*, i.e. cases where the large delay of one cell can significantly affect the delay of many other cells:

- A contagion *within* flows, e.g. when a large packet is segmented into many cells, and one late cell makes *all other cells* wait.
- A contagion *across* flows, e.g. when a late cell of one flow makes cells of *other* flows wait.

The first type is easy to understand. The paper suggests dealing with it by using *network coding*, so as to not have any irreplaceable cell.

The second type is due to *flow blocking*, as explained below, and is tackled using *hash-based counter schemes*.

Today, switches guarantee that packets belonging to the same *switch flow*, *i.e. arriving at the same switch input and departing from the same switch output*, will depart in the same order as they arrived. For switch vendors as well as for their customers, breaking this *switch-flow order* guarantee is not an option.

To the best of our knowledge, *all* current switch designers provide this guarantee.[1] That, even though no standard requires it, and in fact the IPv4 router standard does not even forbid packet reordering (section 2.2.2 in [9]). Today, it is a significant *customer requirement* that needs to be addressed (it is well known that a major core-router vendor lost market share because of its routers experimenting some limited reordering).

[1]However, since switch internal details are often kept confidential, we could not find a publicly available reference. [8] (in Chapter 4), [7], [6] (in Appendix B) refer to the use of internal sequence numbers to maintain packet order, but none of these detail how the sequence numbers are formed.

TABLE I
AN EXAMPLE OF PACKETS WITH THEIR FLOW IDS AND SWITCH FLOW IDS. PACKETS A THROUGH F BELONG TO THE SAME SWITCH FLOW , I.E. SHARE THE SAME SWITCH INPUT AND SWITCH OUTPUT. PACKET G BELONGS TO A SECOND SWITCH FLOW. IN ADDITION, PACKETS A THROUGH F INCLUDE PACKETS FROM 5 DIFFERENT FLOWS, I.E. HAVE 5 POSSIBLE (SOURCE, DESTINATION) IP ADDRESS PAIRS.

Packet	Source IP	Destination IP	Switch Input	Switch Output	Flow ID	Switch Flow ID
A	192.168.0.1	192.168.0.8	1	i	x	I
B	192.168.0.1	192.168.0.8	1	i	x	I
C	192.168.0.1	192.168.255.0	1	i	y	I
D	192.168.1.64	192.168.0.16	1	i	z	I
E	192.168.255.0	192.168.64.0	1	i	u	I
F	192.168.0.0	192.168.0.16	1	i	v	I
G	192.168.64.9	192.168.128.0	N	N	w	II

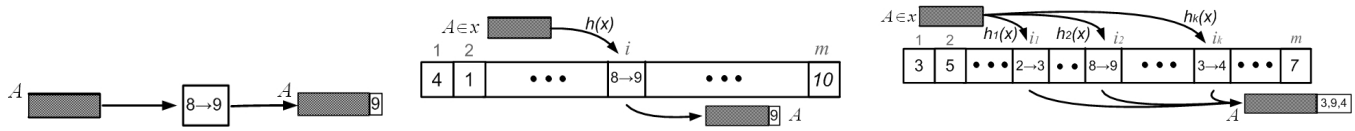
However, this guarantee has been increasingly hard to address in high-end multi-stage switches because of the complex and often-overlooked *flow blocking* interactions. *Within a switch flow, let a flow be the set of all packets that also share the same (source, destination) IP address pair.* Then flow blocking happens when in order to satisfy the switch-flow order guarantee, packets from one flow wait for late packets from *another* flow within the same switch flow.

To illustrate this flow blocking phenomenon, Table I details the packets from Figure 1. Assume that even though packets A through F belong to the same switch flow, they do not all belong to the same flow. Packets A, B belong to flow x (shown with a circle in Figure 1) since they have the same (source, destination) IP pair, packet C belongs to flow y (shown with a pentagon), packet D belongs to flow z (shown with a triangle). Likewise, packet E belongs to flow u and packet F belongs to flow v. An additional packet G belongs to a different switch flow and can independently depart the switch. As previously mentioned, packet A, which arrives to the switch before B and C can depart in order. However, packets D, E and F are out of *switch-flow order*, and therefore need to wait for packets B, C at the output. They can only depart from the switch when B and C arrive, and are *blocked* meanwhile. In particular, note that these packets are blocked by a late packet from a *different* flow: this is *flow blocking*.

C. Our Contributions

In this paper, we *propose solutions for the two different types of the reordering contagion.* We first try to suggest hash-based schemes to separate between flows more accurately and reduce the *inter-flow* blocking, and then suggest a coding scheme to reduce *intra-flow* reordering delay contagion.

First, instead of providing a switch-flow order guarantee, we only provide a *flow order guarantee*, so that packets of the same flow are still maintained in order, but are not constrained with respect to packets of different flows. As mentioned previously, this is compatible with all known standards. We then suggest schemes that use this new order constraint to



(a) *Baseline* scheme (b) *Hashed-Counter* scheme (c) *Multiple-Counter* scheme

Fig. 2. Illustration of the *Baseline* scheme, *Hashed-Counter* scheme and the *Multiple-Counter* scheme. The single flow counter from the *Baseline* scheme is replaced with m hashed flow counters in the *Hashed-Counter* scheme and the *Multiple-Counter* scheme. A packet from a given flow is hashed to a single counter in the *Hashed-Counter* scheme and to k counters in the *Multiple-Counter* scheme.

reduce resequencing delays in the output resequencing buffers. Unlike most previous papers on switch reordering, these schemes *do not change* the internal load-balancing algorithm, and therefore also the internal packet reordering. In other words, our resequencing schemes are end-to-end in the switch, in the sense that they only affect the input and output buffers, and not any element in-between, so as to be transparent to the switch designer internal routing and flow-control algorithms.

Our schemes use various methods to increasingly distinguish between flows and decrease flow blocking. Our first scheme, *Hashed-Counter*, uses a *hash-based counter algorithm*. This scheme extends to switch fabrics the idea suggested in [10] for network processors. It replaces each single flow counter with m hashed flow counters, and therefore effectively replaces flow blocking within large switch flows by reducing it to flow blocking within smaller flow aggregates.

Then, our second scheme, the *Multiple-Counter* scheme, uses the same counter structure, but replaces the single hash function by k hash functions to distinguish even better between flows and therefore reduce flow blocking.

Figure 2 illustrates the *Hashed-Counter* scheme and the *Multiple-Counter* scheme in comparison with a simple counter-based scheme called the *Baseline* scheme. The single counter is replaced with an array of counters in the *Hashed-Counter* and the *Multiple-Counter* schemes. While a packet from a given flow is hashed to a single counter in the *Hashed-Counter* scheme, it is hashed to k counters in the *Multiple-Counter* scheme.

Finally, our last scheme, B_h *Multiple-Counter* attempts to reduce flow blocking even further by using variable-increment counters based on B_h sequences. All these schemes effectively attempt to reduce hashing collisions between different flows within the same switch flow. Note that *in the worst case*, even if all hashes collide, all these scheme guarantee that *they will not perform worse* than the currently common scheme, which uses the same counter for all flows and therefore has the worst flow blocking within any switch flow.

However, if a packet is delayed in a long queue, these counter-based schemes cannot prevent it from affecting many packets within its flow. Therefore, we suggest to *use network coding to reduce reordering delay*. We introduce several possible network coding schemes and discuss their effects on reordering delays. In particular, we show the existence of a time-constrained network coding that is not necessarily related to channel capacity optimality.

Finally, using simulations, we show how the schemes can significantly reduce the total resequencing delay, and analyze the impact of architectural variables on the switch performance. For instance, resequencing delays are reduced by up to an order of magnitude using real-life traces and a real hashing function.

Note that since they do not affect routing, our suggested schemes are general and can apply to a variety of previously-studied multi-stage switch architectures, including Clos, PPS (Parallel Packet Switch), and load-balanced switch architectures [2], [11]–[14]. They can also apply to fat-tree data center topologies with seven stages [15]–[17], and more generally to any network topology with inputs, outputs, and load-balancing with reordering in-between. While we do not expand these for the sake of readability, we believe that our schemes can decrease resequencing delay in all these potential architectures. (For instance, to avoid reordering, data center links are currently oversubscribed by factors of 1:5 or more [15]–[17]. If reordering did not incur such a high resequencing delay, it may be easier to efficiently load-balance packets and fully utilize link capacities.)

D. Related Work

Resequencing schemes for switches are usually divided into two main categories. First, *counter-based schemes*, which rely on sequence numbers. For instance, Turner [18] describes an implementation of a counter-based scheme that corresponds to the *Baseline* scheme, as described later.

Second, *timestamp-based schemes*, which rely on timestamps deriving from a common clock. Henrion [19] introduces such a scheme with a fixed time threshold. Turner [20] presents an adaptive timestamp-based scheme with congestion-based dyamic thresholds. However, while timestamp-based schemes can be simpler to implement, their delays can become prohibitive when internal delays are highly variable, as explained in [20], because most packets experience a worst-case delay. Therefore, we restrict this paper to counter-based schemes.

Several schemes for *load-balanced switches* [12], [13] attempt to prevent *any* reordering within the switch. [2], [4], [21] provide an overview of such schemes. In particular, in the AFBR (Application Flow-Based Routing) as well as in [5], [6] packets belonging to the same hashed flow, are forwarded through the same route, thus preventing resequencing but also changing the flow paths and obtaining low throughput in the worst case. For instance, if there are a few large flows, the achieved balancing is only partial.

Resequencing schemes have also been considered in *network processors*. Wu *et al.* [22] describe a hardware mechanism in each flow gets its own counter. The mechanism remembers all previous flows and sequentially adds new flows

to the list of chains. It then matches a packet from an existing flow with the right SRAM (Static Random-Access Memory) entry using a TCAM (Ternary Content-Addressable Memory) lookup. Meitinger *et al.* [10] are the first to suggest the use of hashing for mapping flows into counters in network processors in order to reduce reordering. They further discuss the tradeoff between the large number of counters and the possible collisions. However, all these works on network processors only consider a single input and a single output, and therefore do not consider the complexity introduced by the N^2 switch flows. In addition, the load-balancing might be contained in network processors with stateful algorithms, while it is not in switches.

Packet reordering is of course also studied in many additional networking contexts, such as the parallel download of multiple parts of a given file [23], or in ARQ (Automatic Repeat reQuest) protocols [24]. Recently, Leith *et al.* [25] considered the problem of encoding packets with independent probabilistic delays such that the probability that a packet can be decoded within a given decoding deadline is maximized.

II. The Hashed-Counter Scheme

A. Background

A commonly used scheme for preserving *switch flow* order is to keep a sequence-number counter for each *switch flow*. In this scheme, denoted as the *Baseline* scheme, each packet arriving at a switch input and destined to a switch output is assigned the corresponding sequence number. As illustrated in Figure 3(a), in each switch input, we keep N counters, one for each switch output. Then, the switch output simply sends packets from this switch input according to their sequence number. Referring to the example of Table I, all the first six packets A through F share the same counter as they belong to the same *switch flow*. Assume that packet A gets sequence number 1, B gets number 2, ..., and F gets number 6. Then packet A can depart without waiting because its sequence number is the smallest. Packet D, E and F need to wait for packets B and C. In the end, the departure order is A, B, C, D, E, F. Likewise, the packet G uses a different counter since it belongs to a second switch flow.

It would seem natural to similarly preserve *flow* order by keeping a counter for each *flow*. However, the potential number of 2^{32+32} (source, destination) IPv4 flows going through a high-performance switch is too large to keep a counter for each.

We could also devise a solution in which counters would only be kept for the most recent flows. But such a solution might be complex to maintain and not worth this cost and complexity. For instance, a 10 Gbps line with an average 400 B packet size would have to keep up to 3 million flows for the last second. If each flow takes $32 + 32$ bits to store the (source, destination) IPv4 addresses and 10 bits for the counter, we would need more than 200 Mb of memory, thus requiring expensive off-chip DRAM (Dynamic Random-Access Memory).

Instead, the following algorithms rely on *hashing* to reduce the number of needed counters by several orders of magnitude, in exchange for a small collision probability.

B. Scheme Description

Figure 4(a) illustrates how the *Hashed-Counter* scheme is implemented in the input port. N arrays of packet counters exist in each input port. For a given output port, the *Hashed-Counter* scheme uses an array of m counters, instead of a single counter for the *Baseline* scheme.

As shown in Figure 4(b), each incoming packet is hashed to a specific counter based on its flow source and destination IP addresses. The counter value is then incremented, and the packet is assigned this value as its sequence number. For instance, a packet A belonging to flow x is hashed to counter $i = h(x)$, and it is assigned sequence number $8 + 1 = 9$. This sequence number 9 is inserted into A, which is forwarded to the middle switch element.

At the output resequencing buffer, the same hash function is used as well. Therefore, it will yield the same counter index. All packets hashing to the same counter will then leave in order, as indicated by their sequence numbers.

Since the hash function is kept constant, all packets of the same flow always use the same counter. Therefore, the switch is *flow order preserving*.

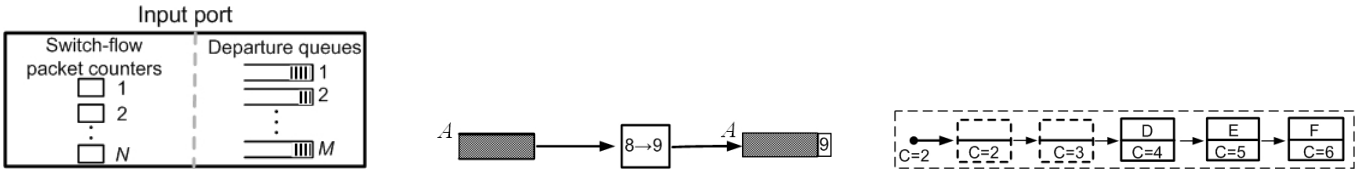
Note that the *Baseline* scheme is a private case of the *Hashed-Counter* scheme for $m = 1$. Intuitively, the *Hashed-Counter* scheme splits all the flows that are part of a switch flow into m sets of flows, and keeps the order within each set. Therefore, a late packet will only delay packets within its set, and not affect the packets of the other $m-1$ sets. Consider the example in Table I again, and assume $m = 4$. Packet A, B, E of flows x, u, and packet C, F of flows y, v are hashed to two different counters, the forth and the second, respectively. Likewise, packet D of flow z is hashed to the third counter. At the output, packet E still needs to be buffered to wait for packet B and packet F for packet C, as in the *Baseline* scheme. However, packet D which is the first packet that uses the third counter, does not necessarily need to be buffered till any other packets's arrival. Thus, it can depart from the switch right after packet A. Therefore, the packet departure order is different from their arrival order and only packets E, F are blocked by the delayed packets B and C.

C. Output Resequencing Buffer

We now want to illustrate the operation of the output resequencing buffer in the two schemes, i.e., *Baseline* and *Hashed-Counter* scheme, by considering the example above.

1) Baseline *Scheme:* As shown in Figure 3(c), the *Baseline* scheme is easy to implement in our example. All packets are kept in a single linked list. When a packet arrives and is the first one in the linked list, it is ready to depart. Since packet A has departed, the next expected packet B has a counter value of 2. Placeholders are used in the linked list for the missing packets B and C. (Of course, real implementations might be optimized by mixing linked lists and arrays and skipping placeholders, but this is beyond the scope of this paper.)

2) Hashed-Counter *Scheme:* Figure 4(c) illustrates how the *Hashed-Counter* scheme is implemented using m separate linked lists. Each packet is hashed into its corresponding linked

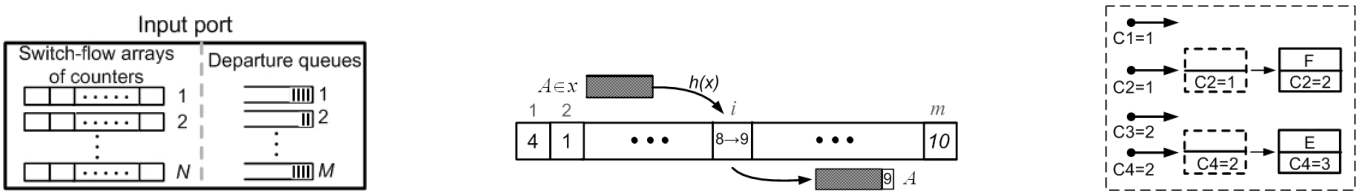


(a) Input port structure, closeup view of Figure 1.

(b) Counter illustration for packet A from flow x.

(c) Output resequencing structure.

Fig. 3. *Baseline* scheme



(a) Input port structure, closeup view of Figure 1.

(b) Counter illustration for packet A from flow x.

(c) Output resequencing structure.

Fig. 4. *Hashed-Counter* scheme

list depending on its flow. When a packet arrives and is the first of its linked list, it is ready to depart.

In this example, flows y and v hash to the second linked list, with a placeholder for delayed packet C and with packet F. Flow z hashes to the third linked list. As the counter value of D is the expected, D leaves immediately. Finally, flows x and u hash to the last linked list, with a placeholder for delayed packet B and with packet E.

III. The Multiple-Counter Scheme

A. Scheme Description

While the *Hashed-Counter* scheme splits the flows into m sets using m counters, we would like to use these counters even more efficiently and split the flows into more sets. We suggest to use several hash functions, while making sure that the order is still preserved.

The implementation of the *Multiple-Counter* scheme is illustrated in Figure 5(a), it is schematically the same as that of the *Hashed-Counter* scheme. As shown in Figure 5(b), the *Multiple-Counter* scheme also keeps an array of m counters for each pair of input and output ports. However, each incoming packet is now hashed into k different counters using k different hash functions of the flow ID (the case $k = 1$ corresponds of course to the previous *Hashed-Counter* scheme). All the k counter values are incremented and sent with the packet. In this example, in the input port, the packet A belonging to flow x is hashed to counters i_1, i_2, i_3 using hash functions h_1, h_2, h_3 and is assigned the sequence numbers $3, 9, 4$, respectively. The sequence numbers are inserted to A, which then is forwarded to the middle switch element.

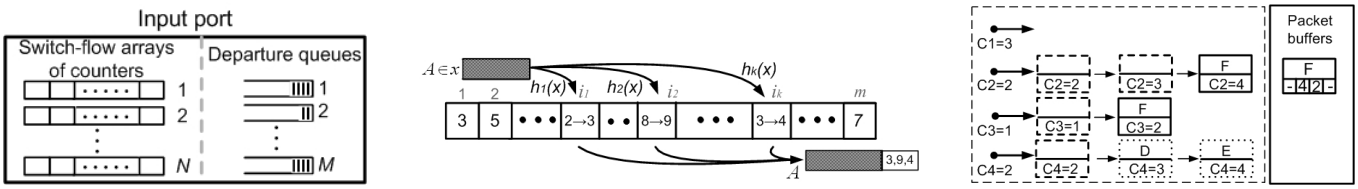
When packet A arrives at the output resequencing buffer, it now needs to check the same k counters. In the case that its counter value is the next expected one in *at least* one of its k counters, we can deduce that P is the earliest packet of its flow, and therefore that it can be released. This is because any earlier packet P' of the same flow would have hashed to the same k counters, and therefore would have had smaller counter values in all counters. If the counter value of P is the next expected one, it necessarily implies that P' has already departed.

The *Multiple-Counter* scheme ensures that the switch is *flow order preserving*. We now describe in greater detail the resequencing buffer implementation.

B. Output Resequencing Buffer

As shown in Figure 5(c), the implementation of the *Multiple-Counter* scheme is similar to that of the *Hashed-Counter* scheme. However, there is a small tweak: each packet actually belongs to several linked lists. Therefore, we only represent packet pointers, while the real packets sit in the packet buffer. Each arriving packet is hashed into its k corresponding linked lists and places pointers in each. When a packet arrives and is the first of *at least one* of its linked lists, it is ready to depart.

We use $k = 2$ and refer again to the example in Table I with additional assumptions. Specifically, we assume that flow x (with A and B) uses counters $\{C2, C4\}$, flow y (with C) uses $\{C2, C3\}$, flow z (with D) uses $\{C1, C4\}$, flow u (with E) uses also $\{C1, C4\}$, and finally flow v (with F) uses $\{C2, C3\}$. Since D is the first packet in the linked list of $C1$, it can leave immediately after its arrival. Later, when E arrives, its counter value for $C1$ is the next expected and therefore it can also leave immediately. Meanwhile, in the list of $C4$, packets D and E are temporally kept with a special mark. When the missing packet B in the list departs, the link will be updated and the expected counter value then would be set to 5. When F arrives, its counter value for $C2, C3$ are 4 and 2 while they expect a packet of counter value 2 and 1, respectively. Therefore, it is not the first of any of the two linked lists. It is placed in each, and a placeholder is added for the missing packets of each counter. In addition, the real packet F is inserted in the packet buffer.



(a) Input port structure, closeup view of Figure 1.

(b) Counter illustration for packet A from flow x.

(c) Output resequencing structure.

Fig. 5. *Multiple-Counter* scheme

Note that this implementation could be readily optimized by using doubly-linked lists, and by pointing directly to the packets instead of using copies. Thus, each packet would contain $2k$ pointers, i.e. two pointers per linked list. It could also be optimized by skipping departed packets in the linked list. All these considerations, however, are beyond the scope of this paper.

IV. The B_h *Multiple-Counter* Scheme

A. B_h *sequences*

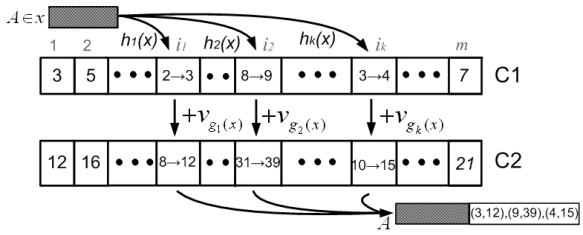


Fig. 6. B_h *Multiple-Counter* scheme in the input, packet A from flow x.

While all previous schemes (*Baseline*, *Hashed-Counter* and *Multiple-Counter*) increment their counters by one upon packet arrival, we now want to introduce *flow-based variable increments* to distinguish between flows even further.

Note how schemes increasingly distinguish between flows and prevent inter-flow blocking: starting from the Baseline scheme, we replace a single flow counter by m hashed flow counters to obtain the Hashed-Counter Scheme. Then we replace a single hash function by k hash functions to obtain the *Multiple-Counter* Scheme. Last, we now distinguish between flows by changing the counter increments, as described below.

To do so, we introduce the B_h *Multiple-Counter* scheme. In this scheme, we keep an array of m *pairs of counters* in each input for each output, as illustrated in Figure 6. The scheme is based on B_h sequences [26]. Intuitively, a B_h sequence is a set of integers with the property that for any $h' \leq h$, all the sums of h' elements from the set are distinct. Therefore, given a sum of h' elements, we can determine whether an element of the B_h *sequence* is a part of the sum. Recently, it was suggested to use B_h sequences in another network application [27].

Definition 1 (B_h Sequence): Let $D = \{v_1, v_2, ..., v_\ell\} \subseteq \mathbb{N}^*$ be a sequence of positive integers. Then D is a B_h *sequence* iff all the sums $v_{i_1} + v_{i_2} + \cdots + v_{i_h}$ with $1 \leq i_1 \leq \cdots \leq i_h \leq \ell$ are distinct, i.e. all the sums of exactly h elements from D have different values.

It is easy to see that a B_h sequence has the following property:

Observation 1: If $D = \{v_1, v_2, ..., v_\ell\}$ is a B_h *sequence* then all the sums $v_{i_1} + v_{i_2} + \cdots + v_{i_{h'}}$ with $1 \leq i_1 \leq \cdots \leq i_{h'} \leq \ell$ for $h' \in [1, h]$ are distinct as well.

Example 1: Let $D = \{v_1, v_2, v_3, v_4\} = \{1, 2, 5, 7\} \subseteq \mathbb{N}^*$.

We can see that all the 10 sums of 2 elements of D are distinct: $1{+}1 = 2, 1{+}2 = 3, 1{+}5 = 6, 1{+}7 = 8, 2{+}2 = 4, 2{+}5 = 7, 2{+}7 = 9, 5{+}5 = 10, 5{+}7 = 12, 7{+}7 = 14$. Therefore, D is a B_2 sequence. However, since $1{+}1{+}7 = 9 = 2{+}2{+}5$, D is not a B_3 sequence.

B. Scheme Description

As illustrated in Figure 6, the scheme uses two sets of k hash functions based on the flow ID: first, $\{h_1, \ldots, h_k\}$, with range $\{1, \ldots, m\}$, where the output of each hash function points to a pair of counters. And $\{g_1, \ldots, g_k\}$, with range $\{1, \ldots, \ell\}$, where the output of each hash function corresponds to a variable increment. At each switch input, each incoming packet P is hashed again into k different array entries. using the hash functions $h_1(P), \ldots, h_k(P)$ of the flow ID. At each array entry $h_i(P)$, there is a pair of counters. The first counter with fixed increments, denoted by $c_1(i)$, is incremented by one. The second counter, $c_2(i)$, is incremented by the element $v_{g_i(P)}$ of the B_h sequence D. The values of the k pairs of counters are then sent with the packet.

The counters are initialized as follows. At the switch input, in each pair of counters, both counters are first set to zero. At the output, the first counter in each pair is set to one (according to the value of the corresponding counter of the first packet). The second counter is initialized to zero. This is the expected value of the second counter in the first packet minus the corresponding variable increment.

When this packet P arrives at the output resequencing buffer, it now needs to check the same k pairs of counters. For each pair of counters in $h_i(P)$, let $d_1(i)$ be the difference between the first counter of this packet and its expected value at the output. We also denote by $d_2(i)$ the difference between (a) the second counter value of the packet minus its variable increment, and (b) the expected value as indicated by the second counter at the output. To calculate $d_1(i), d_2(i)$ we consider the counter values at the output before being updated by the current packet P.

As in the *Multiple-Counter* scheme, if the first counter of the packet is the next expected one, i.e. $d_1(i) = 0$, we can deduce that P is the earliest packet of its flow, and therefore that it can be released. If $d_1(i) \in [1, h]$, we also consider the

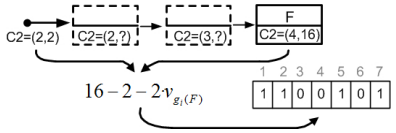


Fig. 7. Output resequencing structure for B_h *Multiple-Counter* scheme.

value of $d_2(i)$. This difference $d_2(i)$ equals the sum of the variable increments of the earlier delayed packets from the current pair of input and output ports. Since these variable increments are based on the B_h sequence D, by definition of the B_h sequence, we can determine whether $d_2(i)$ is composed of $v_{g_i(P)}$. If this is not the case, we can release P even though there are $d_1(i) > 0$ earlier delayed packets that used this array entry. This is because any earlier packet P' of the same flow would have hashed to this pair of counters, and would have incremented this second counter by the same value $v_{g_i(P)}$.

We now consider again the previously suggested example from Table I with additional assumptions on the variable increments. Now, each of $C1, \ldots, C4$ is a pair of counters. Specifically, we are concentrating at $C2$ and assume that flow x (with A, B) increments its variable increment counter of by 2. Flow y (with C) increments it by 7 and flow v (with F) increments it by 5. As in the previous scheme, when F arrives, its fixed increment counter values for $C2, C3$ are 4 and 2 while they expect a packet of counter value 2 and 1, respectively. Therefore, F is not the first of any of the two linked lists. We now consider the values of the pair of counters $C2$. After having been updated by the packets A, B, C and F, the value of the variable increment counter of F is $v_{g_i(A)} + v_{g_i(B)} + v_{g_i(C)} + v_{g_i(F)} = 2 + 2 + 7 + 5 = 16$. Due to the missing packets $B \in x, C \in y$, at the output the value of the corresponding counter before being updated by the current packet F is only 2 (after having been updated only by packet A). Thus, we have that $d_1(2) = 4 - 2 = 2$ and $d_2(2) = (16 - v_{g_i(F)}) - 2 = (16 - 5) - 2 = 9$.

Since $d_1(2) = 2 \leq h$, we can determine that the sum $d_2(2) = 9$ must be composed of exactly two elements of D which are $2 + 7$. Since the relevant variable increment of the flow v is 5, we can deduce that there are not any missing packets of the flow v and the packet F is the earliest packet of its flow and can be released. Since otherwise, we must have that $d_2(2)$ is composed of 5.

C. Output Resequencing Buffer

The implementation of the output resequencing buffer in this scheme is similar to the implementation of the *Multiple-Counter* scheme, besides one change. The decision whether a packet P can be released is based on the values of $d_1(i), d_2(i)$ and $v_{g_i(P)}$. To implement this scheme, we suggest the use of a predetermined two-dimensional binary table based on the B_h sequence D. The value of the table in entry (i, j) equals one iff a sum j can be composed of exactly i elements of D. We can see that the sum $d_2(i)$ can be composed of $v_{g_i(P)}$ and other $d_1(i) - 1$ elements of D iff the sum $d_2(i) - v_{g_i(P)}$ can be composed of exactly $d_1(i) - 1$ elements of D. Therefore, in order to determine if P can be released we access the table at entry $(d_1(i) - 1, d_2(i) - v_{g_i(P)})$. Since $(d_1(i) - 1) \leq (h - 1)$ we must have that $(d_2(i) - v_{g_i(P)}) \leq (h - 1) \cdot \max(D) = (h - 1) \cdot v_\ell$. Thus, the size of this table can be at most $\max(d_1(i) - 1) \cdot \max(d_2(i) - v_{g_i(P)}) \leq (h - 1)^2 \cdot v_\ell$. For $h = 2$ and the B_h sequence $D = \{v_1, v_2, \ldots, v_\ell\} = \{1, 2, 5, 7\}$, we have a total number of $(h - 1)^2 \cdot v_\ell = 1^2 \cdot 7 = 7$ memory bits. As illustrated in Figure 7, in order to determine whether we can release packet F in the example above, we access the table at entry $(d_1(i) - 1, d_2(i) - v_{g_1(F)}) = (2 - 1, 9 - 5) = (1, 4)$. The bit value of zero means that the value of $9 - 5 = 4$ cannot be composed of exactly one element of D and therefore F can be released.

TABLE II
SCHEME COMPARISON

Scheme	Collision Probability	Memory Size/input (*counters*)	Packet overhead Size (*counters*)	Packet processing complexity in re-sequencing buffer
Baseline	1	N	1	1 list
Hashed-Counter	$1/m$	$m \cdot N$	1	1 hash, 1 list
Multiple-Counter	$\frac{1}{\binom{m}{k}}$	$m \cdot N$	k	k parallel $\times$ { 1 hash, 1 list}
B_h *Multiple-Counter*	$\frac{1}{\binom{m}{k} \cdot \ell^k}$	$2m \cdot N$	$2k$	k parallel $\times$ {2 hash, 1 list, 1 table lookup}

V. PERFORMANCE TRADE-OFF OVERVIEW

Table II provides an overview of the properties of the four schemes. It is to give some intuition on the trade-offs involved.

It first shows the *collision probability*, i.e. the probability that two arbitrary packets of a given switch flow use exactly the same counters (with the same variable increments in the B_h *Multiple-Counter* scheme). For simplicity, it assumes that each switch flow consists of an infinity of flows of negligible size and uses uniformly-distributed hash functions. While two packets of a switch flow necessarily collide in the *Baseline* scheme, since they all share the same counter, their collision probability drops down to $1/m$ in the *Hashed-Counter* scheme, because they choose uniformly at random among m counters. In the *Multiple-Counter* scheme, there are $\binom{m}{k}$ different choices of the subset of k counters, and therefore a collision probability of $\frac{1}{\binom{m}{k}}$. Thus, the collision probability is significantly smaller. For the B_h *Multiple-Counter* scheme, collision happens when all the k choices from the B_h sequencing are the same. Consequently, the collision probability is further reduced to $\frac{1}{\binom{m}{k} \cdot \ell^k}$. Note, as discussed in more detail in Section VII-B, that these collision probabilities are not necessarily precisely equal to the probabilities that a given packet is delayed, by a late packet from another flow.

On the other hand, the second column shows that the *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes incur an increase in the number of counters by factors of either m or $2m$. Also, the third column shows that the *Multiple-Counter* and B_h *Multiple-Counter* schemes increase packet overhead. For instance, for $k = 2$, a cell size of 128B and a counter size of 3B. The *Multiple-Counter* scheme adds $6 - 3 = 3\text{B} = 2.3\%$ overhead, and the B_h *Multiple-Counter*

scheme adds $12 - 3 = 9\text{B} = 6.9\%$ overhead. While this overhead is small in front of typical switch speedups (e.g. 2X, and up to to 5X), a switch designer would still need to take it into account.

The last column illustrates schematically the processing complexity needed to insert a packet in the resequencing buffer. The *Baseline* scheme only needs to search through one list to find the packet position. Likewise, the *Hashed-Counter* scheme finds the correct list using a single hash function, then goes through the list. However, the *Multiple-Counter* scheme needs to do so for k lists accessed in parallel. In addition, the B_h *Multiple-Counter* scheme needs another hash function and k parallel table lookups to understand the variable-increment information.

Incidentally, note that these complexity measures might be misleading. For instance, the size of the *Baseline* scheme list is on average *more than* m *times larger* than the lists of the *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes, since their total resequencing buffer size is smaller and they have m lists. Even though finding the largest of k elements on k parallel lists takes more time on average than finding a single one, in practice, the factor m actually makes it significantly easier to reach high access speeds with the *Hashed-Counter* than with the *Baseline* scheme.

VI. Network Coding

A. Coding Against Rare Events

While the counter schemes above can reduce reordering delay, the (*worst-case*) total packet delay is necessarily lower-bounded by the (*worst-case*) queueing delay. This is because the total delay of a packet is composed of its queueing delay and its reordering delay. If the first packet in a flow of 100 packets is delayed in an extremely long queue, then all the other packets will have to wait for it and suffer as well. Therefore, *reordering delay is vulnerable to rare events*.

To solve this problem, we suggest to consider an intriguing idea: *using network coding to reduce reordering delay*. While network coding has been often used in the past, it has mainly been destined to address packet losses, and reordering has often only been a minor side effect [28], [29]. We suggest here to use it exclusively to reduce reordering delay by addressing the vulnerability to rare events. Interestingly, we will show that in some sense, *the total delay of a packet can be smaller than its queueing delay* — because network coding enabled the switch output to reconstruct and send it before it actually arrived to the output.

Consider the switch architecture, as shown in Figure 1, and assume for simplicity that all packets have the same size. To implement network coding, in each input port, we add one packet buffer per switch flow, i.e. a total of N^2 packet buffers in all input ports. Then, for each switch flow, we keep computing the running XOR (exclusive or) function of the previous packets. After a given number of slots (or packets), we send a redundant protection packet that contains this XOR of the previous packets, and re-initialize the XOR computation.

Figure 8 illustrates an example of use of the protection XOR packet. In the input port, the XOR packet X is generated using

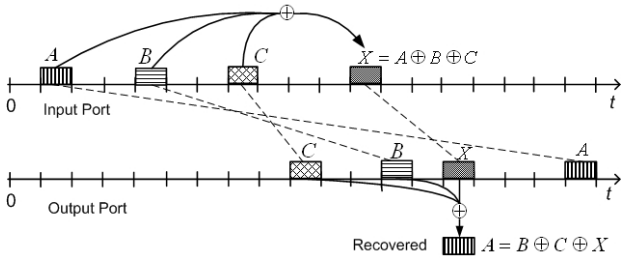


Fig. 8. Network coding example

the previous three packets A, B and C, i.e. $X = A \oplus B \oplus C$. Packet A is delayed by a relatively long time so that it arrives to the output port last. Without XOR packet, packets B and C should wait for A to arrive in order to depart the switch. However, when using network coding, packet A is simply recovered by taking the XOR function of B, C and X, because of the simple identity

$$B \oplus C \oplus X = B \oplus C \oplus A \oplus B \oplus C = A.$$

As we can see, packets A, B and C can depart the switch *before* the original packet A even arrives at the output. Thus, in this example, the XOR packet mainly helps reducing A's queueing delay — and we do not really care about its impact on channel capacity, as in typical network-coding examples.

To further reduce the total delay, more XOR packets can be generated. However, there is no free lunch. The XOR packets will increase the traffic load in the switch, which will result in higher packet queueing delay in the central stage. Therefore, there should exist an optimal point beyond which XORs no longer help.

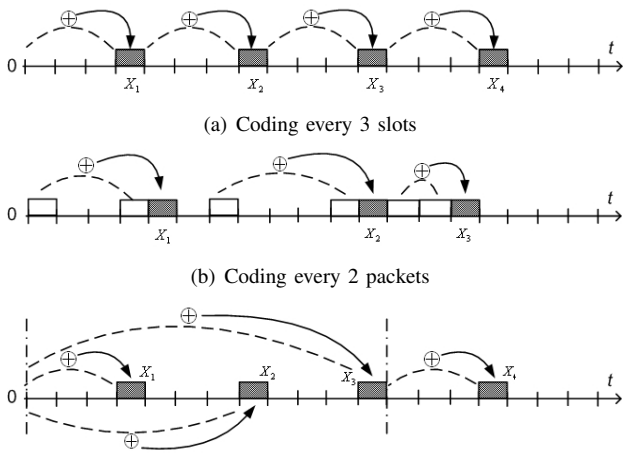
To implement the network coding, in each input port, we keep one packet buffer per each output port. In this packet buffer we keep the cumulative XOR of the packets of the current switch flow which might include packets of several flows. The total number of all packet buffers in all input ports is clearly N^2.

Since for each recovered packet, we would like also to obtain the values of the corresponding counters, the packet buffer should also keep the cumulative XOR of the corresponding counters of the packets, in addition to their data. Thus the required memory size for each pair of input and output port is the sum of the cell size and the total size of the counters assigned to a single packet. This memory overhead is smaller when the fixed size of the cells is smaller.

After a delayed packet is recovered, when it finally arrives at the output it is simply dropped. We can observe that such a situation occurred based on the counters at the switch output.

B. Network Coding Schemes

There are several possible network coding schemes to decide when the protection XOR packets should be sent. First, as shown in Figure 9(a), a simple coding scheme is to generate the XOR packets every H slots by taking the XOR of the packets in the last H slots, where $H = 3$ in the figure. The XOR packet is then inserted following the last of the H slots, and covers the H slots. These $H + 1$ resulting slots make up



(a) Coding every 3 slots

(b) Coding every 2 packets

(c) Cumulative coding every 3 slots in mega-frames of 12 slots

Fig. 9. Network coding schemes

a *frame*. In its header, the protection XOR packet contains the sequence numbers of the first and last packets in the frame, so that the output will be able to know what packet it is supposed to protect.

However, it can practically be further improved in two directions. First, if the traffic load of the flow is light, the number of packets in a frame could be low, and the frame could even be empty. Therefore, the contribution of XOR packets does not justify the high relative overhead and additional delay caused by these XORs. It would make more sense to insert protection XOR packets every L packets instead of every H slots. Figure 9(b) illustrates such a scheme with $L = 2$, where the blank boxes stands for the regular packets and dark boxes for protection XOR packets. The scheme overhead is clearly $\frac{L+1}{L} = 1 + \frac{1}{L}$.

Interestingly, another possible improvement to the first scheme could be to use XORs that protect a variable number of slots, e.g. with cumulative coding. For instance, frames could be grouped into large mega-frames. Within each mega-frame, every H slots, the protection XOR packet cumulatively covers not only the last H slots, but the whole time since the start of the mega-frame, excluding other XOR packets. Therefore, there is a global coding instead of a mere local coding. Figure 9(c) illustrates this cumulative coding scheme with a mega-frame consisting of three frames, each frame having $H = 3$ regular slots and one protection slot.

This last scheme is interesting in that it illustrates an interesting coding tradeoff. On the one hand, we would like the coding to be efficient, so in some sense we could like to wait as much as possible until the end of the mega-frame and then use several protection packets, e.g. using a simple Reed-Solomon or Hamming code with fixed packet dependencies. However, if we only release the protection packets at the end of the mega-frame, the coding becomes useless, because the protected packets will probably have already arrived at the output. Therefore, the protection scheme needs some form of *time-constrained coding*, in the sense that *the protection packets need to be close to the packets they protect.* That's why the schemes displayed above are more effective for reordering delay than more complex schemes that might be closer to the Shannon capacity bounds.

VII. Performance Analysis

A. Delay Models

We now want to model the performance of the schemes under different delay models. We assume a simple Bernoulli i.i.d. in a slotted time, so that the probability of having a packet arrival for a given switch flow at a given slot is equal to p. We analyze the performance of the schemes based on several delay models for the queueing delay T_Q, which is experienced by each packet in the middle switch elements.

First, we will consider a Rare-Event delay model in which T_Q is either some large delay T with probability ϵ, or 0 with probability $1 - \epsilon$. This delay distribution models the impact of low-probability events in which some packets experience very large delays, and these delays impact other packets. For instance, this could be the case of a middle element with a significant temporary congestion. It could also model a failure probability of ϵ for middle elements, with a timeout value T at the output resequencing buffer and a negligible queueing time. After time T, an absent packet is declared lost, and the following packets can be released.

We then consider a General delay model with an arbitrary cumulative distribution function F of the queueing delay T_Q, so that $F_Q(i) = \Pr(T_Q \leq i)$. Then, we apply the analysis to a geometric delay model, such that the delay is geometrically distributed, i.e. $F_Q(i) = \Pr(T_Q \leq i) = 1 - \rho^{(i+1)}$. The Geometric delay model will help us get some intuition on the reordering delay in a switch with load ρ.

In these models, we only take into account the queueing delay T_Q in the middle elements and the resequencing delay T_{RS} in the resequencing buffer. Therefore, the total delay is $T_T = T_Q + T_{RS}$. We neglect the propagation delays, as well as the additional queueing delays in the inputs and outputs. We also assume uniformly-distributed hash functions. For the sake of readability, the performance of the B_h *Multiple-Counter* scheme is brought forward here only for part of the delay models.

B. Delay Distributions

1) Rare-Event delay model: We now want to analyze the *Baseline*, *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes. Note that the performances of the *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes depend on the flow size distribution. For instance, if a switch flow consists of a single large flow, then there is no point in adding counters to distinguish between flows. We have also developed a full model based on the flow sizes. However, for the sake of readability, we will only present below the much simpler case in which all flows have negligible size.

In Delay Model 1, T_Q has one of two possible values. It is equal to 0 w.p. $1 - \epsilon$, and to T w.p. ϵ. The queueing delay is chosen at random independently for each packet. If $T_Q = T$, we say that the packet is delayed. Using this delay model, the total delay $T_T = T_Q + T_{RS}$ has an upper bound of T. The reason is that the queueing delay is either 0 or T.

Therefore, T time slots after its arrival at the switch input, it is guaranteed that a packet arrives to the output. In addition, due to the upper bound of T on the queueing delay, all its earlier packets are also guaranteed to reach the output no later than that time. More generally, the higher T_Q the smaller T_{RS}, and in particular in this delay model if $T_Q = T$ then necessarily $T_{RS} = 0$.

We present the CDF (Cumulative Distribution Function) function of T_T for the different counting schemes. To calculate this function, we have to know the probability of the arrival of another packet of the same flow in the current input-output pair, in each of the previous $T-1$ time slots. For simplicity, we assume that this probability, denoted by p_s, is independent in the time difference of the two packets. According to the assumption that all flows have negligible size and packets are arbitrarily assigned a flow from the possible set of flows, we have that $p_s \approx 0$.

Since the total arrival for this pair is Bernoulli distributed with probability p, we deduce that the probability of an arrival of a packet of a different flow is $p_d = p - p_s \approx p$.

The first theorem is about the Rare-Event delay model. As mentioned earlier, in this delay model, the worst-case delay is T, and therefore we always have $\Pr(T_T \leq T) = 1$.

Theorem 1 (Rare-Event delay model): (i) Using the *Baseline* scheme, for $i \in [0, T-1]$,

$$\Pr(T_T \leq i) = (1-\epsilon) \cdot (1-p\epsilon)^{(T-i-1)}. \quad (1)$$

(ii) Using the *Hashed-Counter* scheme, for $i \in [0, T-1]$,

$$\Pr(T_T \leq i) = (1-\epsilon) \cdot (1-\frac{p}{m} \cdot \epsilon)^{(T-i-1)}. \quad (2)$$

(iii) Using the *Multiple-Counter* scheme, for $i \in [0, T-1]$,

$$\Pr(T_T \leq i) = (1-\epsilon) \cdot [1-(1-p_k^{T-i-1})^k] \quad (3)$$

with $p_k = (1 - p \cdot \epsilon \cdot \frac{k}{m})$.
(iv) Using the B_h *Multiple-Counter* scheme, for $i \in [0, T-1]$,

$$\Pr(T_T \leq i) = (1-\epsilon) \cdot [1-(1-p_{h,k,T-i-1})^k] \quad (4)$$

with $p_{h,k,t} = \sum_{j=0}^{h} \binom{t}{j} (p\epsilon\frac{k}{m} \cdot \frac{\ell-1}{\ell})^j (1-p\epsilon\frac{k}{m})^{(t-j)}$.

Proof: (i) The probability for a given packet *not* to experience a queueing delay of T is $1-\epsilon$. In that case, its reordering delay will be at most i if it doesn't need to wait for earlier delayed packets beyond i slots. Assume that our packet arrived at time t. An earlier packet that arrives at time $t-(T-i)+1$ and is delayed by T slots will arrive at time $t-(T-i)+1+T = t+(i+1)$, causing our packet to wait for $i+1$ slots and therefore miss the time constraint of i. Therefore, to reach this time constraint, any of the $T-i-1$ slots between time $t-(T-i)+1$ and time $t-1$ (included) should either not receive a packet, or not encounter delay. The probability of this event occurring is $(1-p\epsilon)^{(T-i-1)}$.
(ii) This is the same result as above, given a uniformly-distributed hash function and therefore a probability $1/m$ of having another packet share the same counter.
(iii) In the *Multiple-Counter* scheme, a packet can leave the switch if *at least one* of its counters is in order. Therefore, the reordering delay exceeds i if the reordering delay in *all* counters exceeds i, hence the exponent of k. Further, a given counter is shared with a given other packet with a probability of k/m, since this other packet uses k counters out of m. The remainder of the formula is then the same as above.
(iv) In the B_h *Multiple-Counter* scheme, a packet can leave the switch if in *at least one* of its counters there are *at most* h *earlier missing packets* and each of them does not use the same value from the B_h sequence D as the current packet. As explained in the previous scheme, in a time slot we have a delayed packet that uses a specific counter w.p. $p\epsilon\frac{k}{m}$. Since $|D| = \ell$, a specific element of D is not used w.p. $\frac{\ell-1}{\ell}$ as above. ■

2) General delay model: In Delay Model 2, T_Q is distributed according to a general distribution function F, which holds $F_Q(i) = P(T_Q \leq i)$. For example, this delay might be distributed as the distribution of the delay of Geom/Geom/1 queue with Bernoulli distributed arrival with probability p and Bernoulli distributed service with probability q, in each time slot. Using this delay model, the resequencing delay T_{RS}, and the total delay $T_T = T_Q + T_{RS}$ are not bounded from above. Let denote by T_Q^j, the queueing delay of the packet that entered the system at time $t = j$. We also denote $F_Q(i) = P(T_Q \leq i)$. Here, a packet that enters its input port in time $t = t_0$, waits (at most) for $T_T = i$ if this packet and all the other previous packets with the same hashing values that appeared in their input node, arrive to their output node no later than time $t = t_0 + T_T = t_0 + i$.

The next two theorems provide exact models of the performance of schemes given a General delay model and a Geometric delay model.

Theorem 2 (General delay model): (i) Using the *Baseline* scheme,

$$\Pr(T_T \leq i) = F_Q(i) \cdot \prod_{j=1}^{\infty} (1 - p \cdot (1 - F_Q(i+j))). \quad (5)$$

(ii) Using the *Hashed-Counter* scheme,

$$\Pr(T_T \leq i) = F_Q(i) \cdot \prod_{j=1}^{\infty} \Big(1 - \frac{p}{m} \cdot (1 - F_Q(i+j))\Big). \quad (6)$$

(iii) Using the *Multiple-Counter* scheme,

$$\Pr(T_T \leq i) = F_Q(i) \cdot (1 - (1 - p_{k,i})^k), \quad (7)$$

with $p_{k,i} = \prod_{j=1}^{\infty} \big(1 - p \cdot \frac{k}{m} \cdot (1 - F_Q(i+j))\big)$.

Proof: (i) In the General delay model, a packet arriving at time t experiences a total delay of at most i iff it satisfies two independent conditions. First, its queueing delay is at most i, w.p. (with probability) $F_Q(i)$. Second, none of the previous packets have been delayed beyond time $t+i$, and therefore no earlier packets prevent it from leaving. Since an earlier packet arrives at slot $(t-j)$ w.p. p, and in that case is only delayed beyond $t+i$ w.p. $(1-F_Q(i+j))$, the result follows by multiplying all the probabilities that there is no late packet from slot $(t-j)$ over all such possible slots.
(ii) The result follows again directly from above when considering a single counter out of m.
(iii) The proof is again exactly the same as in the previous theorem, and follows from the previous result. ■

3) Geometric delay model: Now we focus on the performance evaluation under a private case of the General delay model with exponential delay. In this case, we have a queuing delay of $T_Q = i$ w.p. $(1-\rho)\cdot\rho^i$ for $i \geq 0$. Therefore, $F_Q(i) = P(T_Q \leq i) = 1-\rho^{(i+1)}$. The results for this case are summarized in the following theorem.

Theorem 3 (Geometric delay model): (i) Using the *Baseline* scheme,

$$\Pr(T_T \leq i) = (1-\rho^{(i+1)}) \cdot \prod_{j=1}^{\infty} \left(1 - p \cdot \rho^{(i+j+1)}\right). \quad (8)$$

(ii) Using the *Hashed-Counter* scheme,

$$\Pr(T_T \leq i) = (1-\rho^{(i+1)}) \prod_{j=1}^{\infty} \left(1 - \frac{p}{m} \cdot \rho^{(i+j+1)}\right). \quad (9)$$

(iii) Using the *Multiple-Counter* scheme,

$$\Pr(T_T \leq i) = (1-\rho^{(i+1)}) \cdot (1-(1-p_{k,i})^k), \quad (10)$$

with $p_{k,i} = \prod_{j=1}^{\infty} \left(1 - p \cdot \frac{k}{m} \cdot \rho^{(i+j+1)}\right)$

Proof: The results follow from the previous theorem using the expression of the geometrically-distributed delay model. ■

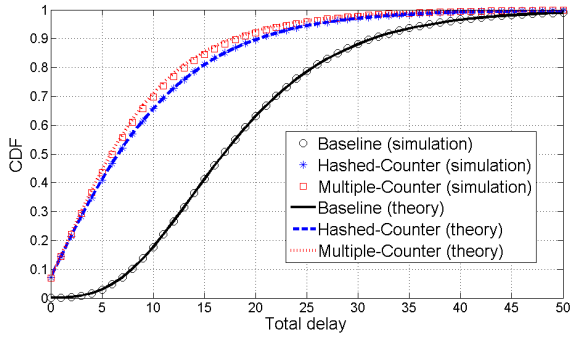
VIII. Simulations

We fully developed from scratch a switch simulator that can run several use-cases, including the geometric delay model, the Clos switch and the data center switch. To better mimic the practical network behavior, we further added a possibility of feeding the flow source with real-life traces [30]. Simulations for each use-case are conducted as described below. Throughout this section, delays are brought in units of time-slots. In the simulations, we rely on real-life 64-bit mix hash functions [31]. In addition, for the evaluation of the B_h *Multiple-Counter* scheme, we use the B_h sequence $D = \{v_1, v_2, \ldots, v_\ell\} = \{1, 4, 8, 13\}$ which is a B_h sequence for $h = 3$.

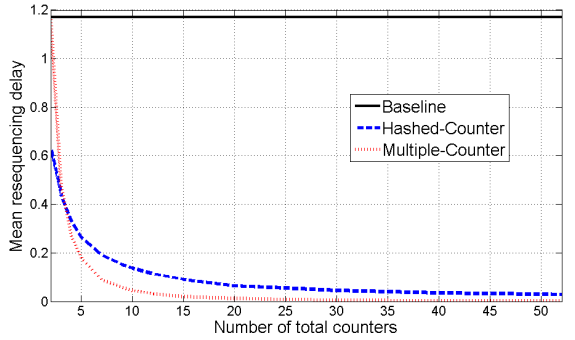
A. Geometric Delay Model Simulations

We run simulations for the geometric delay model discussed in Section VII-B. In the simulation, the number of flows equals 2^{19}, so that the assumption that the probability that two arbitrary packets share the same flow is negligible still holds. We generate the switch flow using Bernoulli i.i.d traffic with arrival rate $p = 0.77$, the parameter of geometric delay $\rho = 0.88$, and the number of total counters $m = 10$. In the *Multiple-Counter* scheme and the B_h *Multiple-Counter* scheme, we use $k = 2$ hash functions.

Figure 10(a) depicts the simulation results, compared with the theoretical results from (8), (9) and (10). The simulation results match the theoretical results quite well. Furthermore, we can see that changing the ordering guarantee from switch-flow order preservation to flow order preservation significantly decreases the average and standard-deviation of both delays. Using the *Hashed-Counter* scheme, the average total delay is drastically reduced from 19.09 to 9.35 time slots, an improvement of 51%. In the *Multiple-Counter* scheme the average total delay is reduced again by additional 8.3% to 8.58 without any additional memory.



(a) CDF of the total delay (in time slots) for the geometric delay model (based on simulation and Theorem 3).



(b) Mean resequencing delay for the geometric delay model as a function of the number of total counters (based on simulation).

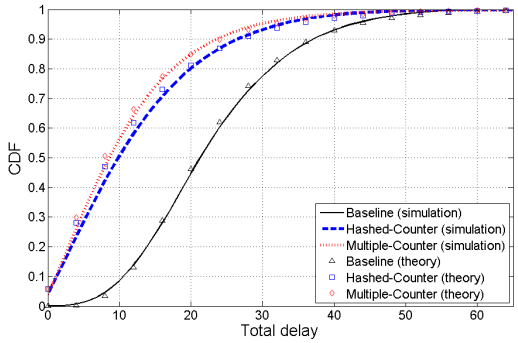
Fig. 10. Geometric delay model

Figure 10(b) shows the average resequencing delay as a function of the number of counters m. The *Hashed-Counter* and the *Multiple-Counter* scheme significantly reduce the re-sequencing delay of the *Baseline* scheme which is not affected of course by the value of m. For instance, for $m = 10$, the average resequencing delay for the *Baseline*, *Hashed-Counter* and *Multiple-Counter* schemes is 11.7315, 2.0312 and 1.2442, respectively. Furthermore, for $m = 80$, the delay of the *Hashed-Counter* scheme is 0.2731 while the *Multiple-Counter* scheme reduces it to only 0.0302.

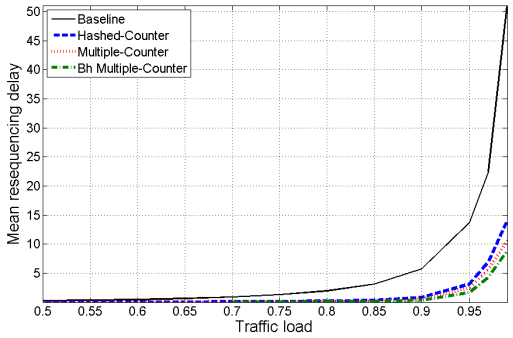
B. Switch Simulations

We now run simulations with the switch structure from Figure 1. Therefore, the delay experienced in the middle switch elements does not follow a specific delay model as above, but is instead incurred by other packets. We always keep $N = 4$ and $M = 8$, yielding $N^2 = 16$ switch flows going through $MN^2 = 128$ different paths. We set the number of total counters to $m = 20$ for the *Hashed-Counter* and *Multiple-Counter* schemes. For the B_h *Multiple-Counter* scheme we set $m = 10$ to account for the larger memory requirements. We also assume a uniform traffic matrix. The presented results are based on simulations.

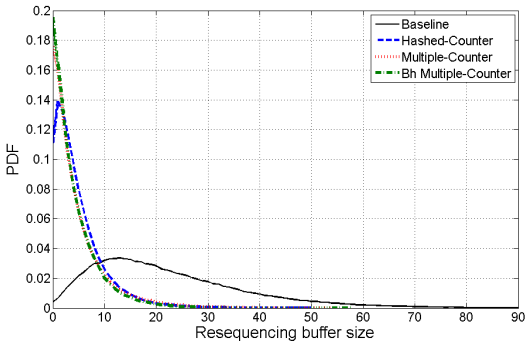
We start by comparing the performance of hash-based counter schemes by using 4096 flows per (input, output) pair, i.e. a total of $4096 \cdot 16 = 65,536$ flows. The total load is



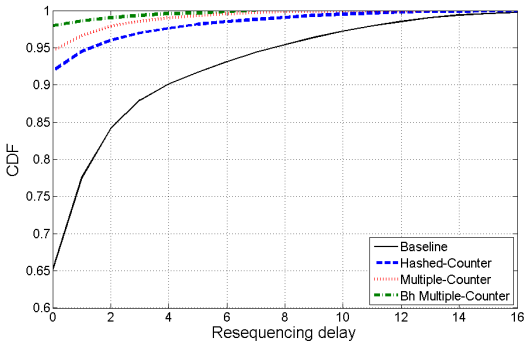
(a) Comparison of the switch to the geometric delay model.



(b) Mean resequencing delay for the switch as a function of the traffic load.



(c) PDF of resequencing buffer size for the switch.



(d) CDF of the resequencing delay for a switch, using a real-life trace.

Fig. 11. Switch simulation results

set to $p = 0.95$. Each flow is generated using Bernoulli i.i.d. traffic of parameter $p/65,536$. In the *Multiple-Counter* and B_h *Multiple-Counter* schemes, we have $k = 2$.

Figure 11(a) compares the results of the switch simulations to the theoretical results from (8), (9) and (10), similarly to Figure 10(a). For the *Baseline* scheme the optimal fit was reached using $\rho = 0.11$, and for the *Hashed-Counter* scheme and *Multiple-Counter* scheme using $\rho = 0.091$. The discrepancy between the theory and the simulation can be explained when we consider that in the switch simulations the assumption that the flows are mice of negligible size does not hold.

Figure 11(b) plots the average resequencing delay in the switch as a function of the traffic load p. As the load increases and the delays become larger, the impact of the *Hashed-Counter* scheme, *Multiple-Counter* scheme and B_h *Multiple-Counter* scheme becomes increasingly significant. The B_h *Multiple-Counter* scheme results in the lowest resequencing delay. For instance, at high traffic load, $p = 0.99$, the *Multiple-Counter* scheme achieves a 24.2% reduction of the resequencing delay obtained by the *Hashed-Counter* scheme while the B_h *Multiple-Counter* scheme further reduces it by additional 16%.

Figure 11(c) shows the PDF (Probability Density Function) of the number of packets, which are blocked from exiting the resequencing buffers for the *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes. The hash-based schemes vastly improve the average number of waiting packets in the *Baseline* scheme i.e, the *Baseline* scheme has an average of 22.47 packets per time slot, while the *Hashed-Counter* scheme has approximately 5. The *Multiple-Counter* scheme reduces this number to 4.6 packets per time slot, and finally the B_h *Multiple-Counter* scheme has only 4.07 packets per time slot.

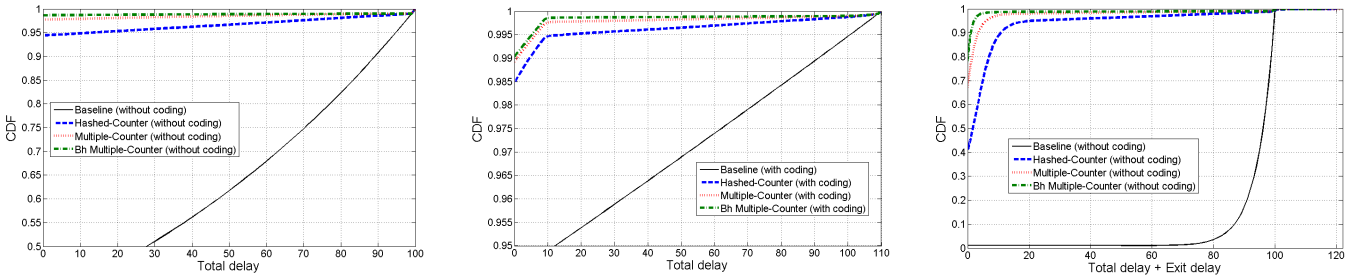
C. Switch Simulations Using Real-Life Traces

We now conduct experiments using real-life traces recorded on a single direction of an OC192 backbone link [30]. We use a real hash function [31] to match each (source, destination) IP-address flow with a given counter bin. In the *Multiple-Counter* and B_h *Multiple-Counter* schemes, we have $k = 2$.

As expected, Figure 11(d) shows how the *Hashed-Counter* scheme drastically reduces the resequencing delay on this real-life traffic, from 1 to 0.1. Again, the reduction in total delay is more modest, from 2.4 to 1.5 time slots.

D. Network Coding Simulations

Figure 12 presents simulation results of the CDF of the total delay for the Rare-Event delay model with the parameters $T = 100$, $\epsilon = 0.01$. We assume a traffic load of $p = 0.95$ and that the flows of the packets are uniformly distributed among a set of 4096 possible flows. The network coding parameter was $L = 10$, i.e. a protection XOR packet was inserted every 10 packets. In order to have a fair comparison, we assumed that



(a) CDF of the total delay without coding.

(b) CDF of the total delay with coding.

(c) CDF of the total delay + exit delay without coding.

Fig. 12. CDF of the total delay and the total delay + exit delay for the rare-event delay model with and without network coding. The results are based on simulations.

when the network coding is used and the load is increased by factor $\frac{L+1}{L}$, T is increased by the same factor. Thus, the maximal queueing delay is 110. Figure 12(a) presents the CDF of the total delay without using the network coding and Figure 12(b) shows the CDF with coding.

CDF of 0.9 is achieved in the *Baseline* without coding only for a total delay of 90. For *Baseline* with coding and for all other schemes, this CDF is achieved even for delay of 0. Without network coding, the mean of the total delay for the *Baseline*, *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes was 36.12, 3.33, 1.53 and 1.14, respectively. Using the network coding, the mean delay was significantly reduced to 3.30, 0.41, 0.23 and 0.17. An improvement by approximately an order of magnitude.

In the delay models from Section VII and in the simulations so far, we have assumed that a packet can leave the switch *immediately* when the packet is available at the switch output and in order. In practice, in many switches at most one packet might leave the same switch output in a single time slot. Thus a packet that is ready to leave the switch at the same time as a second packet, might experience additional delay. We denote this delay by *exit delay* and assume that the switch gives precedence to packets with earlier arrival time. Figure 12(c) presents the CDF of the sum of the total delay and the exit delay in the Rare-Event delay model (without network coding). The mean of this sum of delays for the *Baseline*, *Hashed-Counter*, *Multiple-Counter* and B_h *Multiple-Counter* schemes was 94.00, 6.64, 2.50 and 1.52, respectively. With network coding the means were reduced to 42.24, 1.30, 0.54 and 0.37, respectively. (We do not present the full CDF for this case with network coding due to space limits.)

E. Data Center Simulations Using Real-Life Traces

We also perform simulations on a 5-stage data center structure described in [17]. The input ports are fed with a real-life trace as in Section VIII-C. The total number of counters in the *Hashed-Counter* scheme and *Multiple-Counter* scheme is $m = 20$ while for the B_h *Multiple-Counter* scheme we have $m = 10$. In the *Multiple-Counter* scheme and B_h *Multiple-Counter* scheme, the number of multiple counters is $k = 2$.

Figure 13 illustrates the simulation results of the resequencing delay. As expected, *Hashed-Counter* scheme noticeably reduces the resequencing delay, from 16.3 to 6.73 time slots. In this data-center simulation, the improvement from the *Hashed-Counter* scheme to the *Multiple-Counter* scheme is more significant. The average resequencing delay is decreased from 6.73 to 3.30 time slots (a 51.0% improvement). The reason is that as the packet goes through more stages of switch elements, the variance of the queueing delay is larger and the reordering caused by other flows becomes more severe. This delay can be improved furthermore by the *Multiple-Counter* scheme. As in the previous simulations, the best results are obtained by using the B_h *Multiple-Counter* scheme, for which the resequencing delay is only 1.84 time slots.

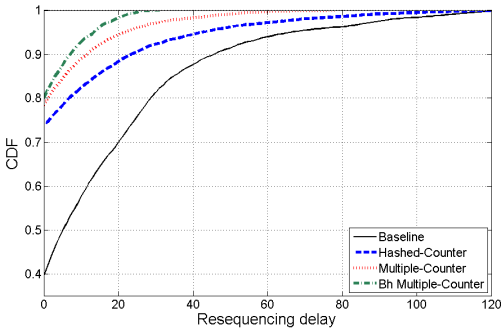


Fig. 13. CDF of the resequencing delay for a data center, using a real-life trace (based on simulation).

IX. Conclusion

In this paper, we provided schemes to deal with packet reordering, an emerging key problem in next-generation switch designs. We argued that current packet order requirements for switches are too stringent, and suggested only requiring flow order preservation instead of switch-flow order preservation.

We then suggested several schemes to reduce reordering. We showed that hash-based counter schemes can help prevent inter-flow blocking. Then, we also suggested schemes based on network coding, which are useful against rare events with high queueing delay, and identified a time-constrained coding problem. We also pointed out an inherent reordering delay unfairness between elephants and mice, and suggested several mechanisms to correct this unfairness. We finally demonstrated in simulations reordering delay gains by factors of up to an order of magnitude.

In future work, we intend to further investigate the optimality of our schemes. We intend to find whether there are fundamental lower bounds to the average delay caused by reordering in a switch, given any possible scheme.

X. Acknowledgment

We would like to thank Alex Shpiner for his helpful suggestions. This work was partly supported by the European Research Council Starting Grant No. 210389, by the Jacobs-Qualcomm fellowship, by an Intel graduate fellowship, by a Gutwirth Memorial fellowship, by an Intel research grant on Heterogeneous Computing, by the Intel ICRI-CI Center, and by the Hasso Plattner Center for Scalable Computing.

Ori Rottenstreich Ori Rottenstreich received the B.S. degree in computer engineering from the electrical engineering department of the Technion, Haifa, Israel in 2008. He is now pursuing a Ph.D. degree in the same department. He is mainly interested in computer networks as well as in algorithm design and analysis. Ori Rottenstreich is a recipient of the Google Europe Fellowship in Computer Networking, the Andrew Viterbi graduate fellowship, the Jacobs-Qualcomm fellowship, the Intel graduate fellowship and the Gutwirth Memorial fellowship.

Pu Li Pu Li was in the Department of Electrical Engineering, Technion, between November 2009 and July 2010. He is now working as New Technology Introduction Engineer in ASML, Netherlands. Previously, he received the B.S. and M.Sc. from Zhejiang University, China, and Technical University Eindhoven, the Netherlands, in 2007 and 2009, respectively. He conducted multiple research projects in the Chinese University of Hong Kong, TNO Netherlands and Philips Research Europe, in the field of communication network. His research interests include wireless technologies, high-performance switching network and network-on-chip.

Inbal Horev Inbal Horev is an M.Sc student at the department of Mathematics and Computer Sciences at the Weizman Institute of Science in Rehovot, Israel. She holds a B.Sc in both Physics and Electrical Engineering from the Technion - Israel Institute of Technology, Haifa, Israel. Current research fields include Active Learning, Computer Vision and Geometry Processing.

Isaac Keslassy Prof. Isaac Keslassy received the M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, Stanford, CA, in 2000 and 2004, respectively.

He is currently an associate professor in the Electrical Engineering department of the Technion, Haifa, Israel. His recent research interests include the design and analysis of high-performance routers and on-chip networks. The recipient of the ERC Starting Grant, the Yigal Alon Fellowship and the ATS-WD Career Development Chair, he is a senior member of the IEEE and an editor for the IEEE/ACM Transactions on Networking.

Shivkumar Kalyanaraman Shivkumar Kalyanaraman is a Senior Manager at IBM Research - India, Bangalore. Previously, he was a Professor at the Department of Electrical, Computer and Systems Engineering at Rensselaer Polytechnic Institute in Troy, NY. He holds degrees from Indian Institute of Technology, Madras, India (B.Tech), Ohio State University (M.S., Ph.D.) and RPI (Executive MBA). His research in IBM is at the intersection of emerging wireless technologies, smarter energy systems and IBM middleware and systems technologies. He is a Fellow of the IEEE, and ACM Distinguished Scientist.

References

[1] F. Abel *et al.*, "Design issues in next-generation merchant switch fabrics," *IEEE/ACM Trans. Networking*, vol. 15, no. 6, pp. 1603-1615, 2007.

[2] I. Keslassy *et al.*, "Scaling internet routers using optics," in *ACM SIGCOMM*, 2003.

[3] N. Chrysos *et al.*, "End-to-end congestion management for non-blocking multi-stage switching fabrics," in *IEEE/ACM ANCS*, 2010.

[4] I. Keslassy, *The Load-Balanced Router,* VDM Verlag, 2008.

[5] "Cisco Catalyst 6500 documentation." [Online]. Available: http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.2SXF/native/configuration/guide/cef.html

[6] "Juniper - T Series Core Routers Architecture Overview." [Online]. Available: http://www.juniper.net/us/en/local/pdf/whitepapers/2000302-en.pdf

[7] "Cisco CRS-1 Overview." [Online]. Available: http://cseweb.ucsd.edu/ varghese/crs1.ppt

[8] "Cisco CRS Carrier Routing System 16-Slot Line Card Chassis System Description." [Online]. Available: http://www.cisco.com/en/US/docs/routers/crs/crs1/16-slot-lc/system-description/reference/guide/sysdsc.pdf

[9] F. Baker, "RFC 1812: Requirements for IP Version 4 Routers," June 1995. [Online]. Available: http://www.faqs.org/rfcs/rfc1812.html

[10] M. Meitinger *et al.*, "A Hardware Packet Resequencer Unit for Network Processors," in *Architecture of Computing Systems (ARCS)*, 2008.

[11] S. Iyer and N. McKeown, "Analysis of the parallel packet switch architecture," *IEEE/ACM Trans. Networking*, vol. 11, no. 2, pp. 314-324, 2003.

[12] C.-S. Chang, D.-S. Lee and Y.-S. Jou, "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering," *Computer Communications*, vol. 25, no. 6, pp. 611-622, 2002.

[13] C.-S. Chang, D.-S. Lee and C.-M. Lien, "Load balanced Birkhoff-von Neumann switches, part II: multi-stage buffering," *Computer Communications*, vol. 25, no. 6, pp. 623-634, 2002.

[14] W. Shi, M. H. MacGregor and P. Gburzynski, "Load balancing for parallel forwarding," *IEEE/ACM Trans. Networking*, vol. 13, no. 4, pp. 790-801, 2005.

[15] A. G. Greenberg *et al.*, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM*, 2009.

[16] M. Al-Fares, A. Loukissas and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008.

[17] R. N. Mysore *et al.*, "PortLand: a scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM*, 2009.

[18] J. S. Turner, "Resequencing Cells in an ATM Switch," Washington University, Computer Science department, WUCS-91-21, 2/91.

[19] M. Henrion, "Resequencing system for a switching node," U.S. patent #5,127,000, 6/92

[20] J. S. Turner, "Resilient Cell Resequencing in Terabit Routers," in *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2003.

[21] S. Kandula *et al.*, "Dynamic load balancing without packet reordering," *Computer Communication Review*, vol. 37, no. 2, pp. 51-62, 2007.

[22] B. Wu *et al.*, "A Practical Packet Reordering Mechanism with Flow Granularity for Parallelism Exploiting in Network Processors," in *IEEE IPDPS*, 2005.

[23] Y. Nebat and M. Sidi, "Parallel downloads for streaming applications - a resequencing analysis," *Performance Evaluation*, vol. 63, no. 1, 2006.

[24] Y. Xia and D. N. C. Tse, Analysis on packet resequencing for reliable network protocols, *Performance Evaluation*, vol. 61, no. 4, pp. 299-328, 2005.

[25] D. J. Leith and D. Vasudevan, "Coding packets over reordering channels," in *IEEE ISIT*, 2010.

[26] S. W. Graham, "B_h sequences," *Analytic Number Theory*, vol. 1 (Allerton Park, IL, 1995).

[27] O. Rottenstreich, Y. Kanizo and I. Keslassy, "The Variable-Increment counting Bloom filter," in *IEEE Infocom*, 2012.

[28] O. Tickoo *et al.*, "LT-TCP: End-to-End Framework to Improve TCP Performance over Networks with Lossy Channels," in *IEEE/ACM IWQoS*, 2005.

[29] V. Sharma *et al.*, "MPLOT: A Transport Protocol Exploiting Multipath Diversity Using Erasure Codes," in *IEEE Infocom*, 2008.

[30] C. Shannon *et al.*, "CAIDA Anonymized 2008 Internet Trace," http://imdc.datcat.org/collection/.

[31] T. Wang, "Integer Hash Function," http://www.concentric.net/ Ttwang/tech/inthash.htm.