Redefining Switch Reordering

Ori Rottenstreich*, Pu Li*, Inbal Horev*, Isaac Keslassy* and Shivkumar Kalyanaraman†
*Department of Electrical Engineering
Technion - Israel Institute of Technology
Haifa 32000, Israel
Emails: {or@tx, puli@tx, ihorev@tx, isaac@ee}.technion.ac.il
†IBM Research, India
Email: shivkumar-k@in.ibm.com

***Abstract*—Packet reordering has now become one of the most significant bottlenecks in next-generation switch designs. In this paper, we argue that current packet order requirements for switches are too stringent, with little or no reason. Instead of requiring all packets sharing the same switch input and switch output to be ordered, we only require packets that share the same source and destination IP addresses to be ordered. We then exploit this new definition by suggesting several hash-based counter schemes that prevent inter-flow blocking and reduce reordering delay. The schemes are transparent to the routing scheme. We further suggest schemes based on network coding to protect against rare events with high queueing delay. We also point out an inherent reordering delay unfairness between elephants and mice, and introduce several mechanisms to correct this unfairness. 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 a factor of 10 using real-life traces and a real hashing function.**

I. Introduction

A. Flow Blocking

Packet reordering is one of the most significant previously-overlooked problems that now feature among the main bottlenecks to high-end next-generation switch designs.

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 (e.g. [1]–[3]), even though no standard requires it, and in fact the IPv4 router standard does not even forbid packet reordering (section 2.2.2 in [4]).

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.

Figure 1 illustrates this flow blocking phenomenon in a typical $N \times N$ multi-stage switch architecture. The first stage of the switch consists of N input ports. The input ports are connected using a first mesh to the second stage of M middle elements. Then, these middle elements are connected using a second mesh to the third and last stage of N output ports. Whenever a packet arrives to a switch input port, it first increments its switch-flow counter and copies the counter value it in its header. The packet is then load-balanced (either uniformly-at-random or using round-robin) to one of the M departure queues, before leaving for the corresponding middle element. Likewise, in the middle element, it is placed in the departure queue that corresponds to its output port destination, and then forwarded to its output port. Last, in its output port, it first waits in its resequencing structure to be resequenced based on its switch-flow counter value, and then joins the departure queue to depart the switch.

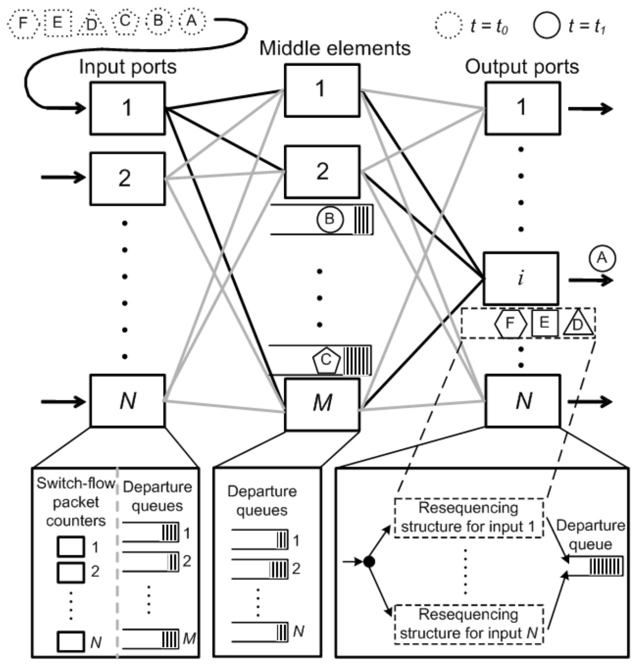


Fig. 1. Switch flow blocking in the switch architecture.

Let us now illustrate flow blocking. As shown in Figure 1, consider an incoming stream of packets A through F that belong to the same switch flow (from input 1 to output i). Within this switch flow, packets A, B belong to flow x (shown with a circle), 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. Assume that the routing algorithm will let the six packets use different middle switch elements, and that packets B and C are blocked in the queue of their middle switch elements which are temporarily congested. Then packet A, which arrived 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 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*.

B. Scaling Bottlenecks

More generally, assume that some middle element experiences a temporary delay of T. For instance, this could be due to temporary congestion (following the sudden arrival of many unicast or a few high-fanout multicast packets), or to different flow-control message propagation latencies on different links, or to a periodic maintenance algorithm. Then due to the input round-robin load-balancing across middle elements, *all* switch flows with at least N packets will have at least one packet that will go through this middle element and be delayed. In addition, because of flow blocking, this delayed packet will block *all* the later packets of its switch flow that wait for it at the output, even though they went through different middle elements. Therefore, the switch will nearly behave as if *all* the middle elements experienced this delay of T. Instead of affecting only $\frac{1}{M}^{th}$ of the traffic, it potentially affects all the traffic. As seldom noticed previously, the switch practically experiences a *spreading of the worst-case delay* due to the switch-flow order guarantee.

This worst-case delay spreading caused by the switch-flow order guarantee is at the source of three bottlenecks to scalability: *delay*, *buffer size*, and *implementation complexity*.

First, it clearly increases *switch delays*. It could be argued that high-end switches like Internet backbone switches can allow for reasonable delays, especially when compared to the large backbone link propagation latencies. However, following the recent emergence of the data center switch market, even backbone switch designers design their switches so they could later be easily modified to enter the data center market as well. But data center switches have stringent delay requirements. For instance, a recent benchmark study favored two switches over a third because of a few μs of delay [5]. Designers of next-generation high-end switches are currently adopting these stringent delay requirements. While reasonable resequencing delays were acceptable, they have now become prohibitive.

In addition, even if high resequencing delays are allowed, they can make the designer tasks much harder. For instance, in the example above, assume that the delay of the middle element is $T = 1$ ms. Then the *resequencing buffer size* should allow for about 1 ms of blocked traffic. At next-generation rates of 100 Gbps, the buffer size should be at least 12 MB, which is beyond the buffer sizes of commodity SRAM. This forces designers to use either off-chip DRAM, which limits chip pin bandwidth, or on-chip eDRAM (embedded DRAM) or 1T-SRAM, which can make on-die integration difficult and therefore raise production costs.

Finally, the large buffer size also causes *implementation complexity* issues. The output resequencing structures (shown in Figure 1) are conceptually implemented as linked lists. Longer resequencing buffers imply that designers need to insert cells into longer linked lists at a small constant average time. For instance, assuming 128-Byte cells at 100 Gbps and $T = 1$ ms yields an average insertion time of 10 ns for a worst-case linked list size of $100,000$ cells.

C. Our Contributions

In this paper, we propose to *redefine the ordering constraint* in the resequencing buffer. 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 definition to reduce resequencing delays in the output resequencing buffers, *while keeping packet paths inside the switch unchanged*. 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. To our knowledge, these are the first end-to-end schemes in the literature that can leverage the flow order guarantee definition.

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*. 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, *Multiple-Counter*, 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.

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.

We later provide an overview of the implementation tradeoffs for our suggested schemes.

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 in 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, as well as the possibility of space coding within the switch fabric elements.

Later, we point out that delay-sensitive large and bursty flows suffer more from reordering delay, hence incurring some *reordering unfairness*. We suggest several schemes that favor these flows to correct this unfairness.

We then provide analytical models of the performance of our suggested schemes. These models provide more intuition and help evaluate the tradeoff of reordering delay and unfairness vs. the overhead and complexity of the algorithms.

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 a factor of 10 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, and load-balanced switch architectures [6]–[9]. They can also apply to fat-tree data center topologies with seven stages [10]–[12], and more generally to any network topology with inputs, outputs, and load-balancing with reordering in-between. While we do not expand on these for lack of space, 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 [10]–[12]. If reordering did not incur such a high resequencing delay, it might 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 [13] describes an implementation of a counter-based scheme that corresponds to our *Baseline* scheme.

Second, *timestamp-based schemes*, which rely on timestamps deriving from a common clock. Henrion [14] introduces such a scheme with a fixed time threshold. Turner [15] presents an adaptive timestamp-based scheme with congestion-based dyamic thresholds. However, we restrict this paper to counter-based schemes. While timestamp-based schemes can be simpler to implement, their delays can become prohibitive in switches with stringent delay requirements.

Several schemes for *load-balanced switches* [7], [8] attempt to prevent *any* reordering within the switch. [9], [16] provide an overview of such schemes. In particular, AFBR forwards packets belonging to the same hashed flow through the same route, thus preventing resequencing but also changing the flow paths and obtaining low throughputs in the worst case. Also, UFS groups packets by frames of M packets before slicing them across all middle elements. However, it can also incur high delays to form envelopes. Additional schemes like FOFF only limit the amount of reordering, and can be combined with our suggested schemes [9], [16], [17].

Resequencing schemes have also been considered in *network processors*. Wu *et al.* [18] 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 entry using a TCAM lookup. Further, Meitinger *et al.* [19] suggest the use of hash functions to map flows to counters, using a scheme that is related to the *Multiple-Counter* scheme. 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 discuss the complexity introduced by the N^2 switch flows. In addition, the load-balancing might be contrained 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 [20], [21], or in ARQ protocols [22].

E. Dependence Among Flows

Using the new definition of order preservation, two packets sharing the same switch input and output, yet belonging to different flows, might be reordered.

However, this might be a problem if the two flows are actually interdependent. For instance, we could imagine that a given client first sends a *write* packet to some data center server. Then, it could send a second *read* packet for the same chunk of data to a different server of the same data center, or to a different IP address of the same server.

We believe that such examples are extremely rare and can be neglected. In fact, if these flows were to go through different switch inputs or outputs, they would belong to different switch flows, and therefore their order would already not be preserved under the current definition of switch-flow order preservation.

Nevertheless, if a network administrator wants to keep the dependence among such flows, a simple solution would be to add a small TCAM (Ternary Content Addressable Memory) in each input port. Then, all packets belonging to any of these dependent flows would be simply marked as belonging to the same flow. For instance, all packets belonging to flow *(SrcIP 1, DstIP 1)* and flow *(SrcIP 2, DstIP 2)* would be marked as belonging to the first flow *(SrcIP 1, DstIP 1)*. As a consequence, any flow-order-preserving switch would also preserve the order of these dependent flows.

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. Then, the switch output simply sends packets from this switch input according to their sequence number. Referring to the example of Figure 1, all the six packets share the same counter as they belong to the same *switch flow*. Assume that packet A get 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.

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.

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 3(a) illustrates how the *Hashed-Counter* scheme is implemented in the input port. There are 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 3(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 also used. 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 Figure 1 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 which uses the third counter, does not necessarily need to be buffered till any other packets's arrival. Thus, it can departure the switch right after packet A. Therefore, the packet departure order is different of 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 2(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 of 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 3(c) illustrates how the *Hashed-Counter* scheme is implemented using m separate linked lists. Each packet is hashed into its corresponding linked 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 z hash to the second linked list, with a placeholder for delayed packet C and with packets F. Flows 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

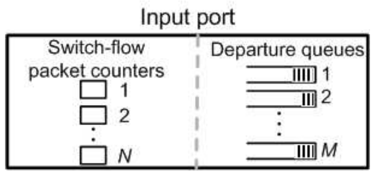
A. Scheme Description

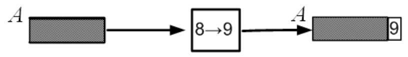
While the *Hashed-Counter* scheme splits the flows into m sets using m counters, we would like to use these counters even more efficiently and split the flows into more sets. We suggest to use several hash functions, while making sure that the order is still preserved.

The implementation of the *Multiple-Counter* scheme is illustrated in Figure 4(a), it is schematically the same as that of the *Hashed-Counter* scheme. As shown in Figure 4(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.


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


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

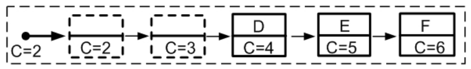
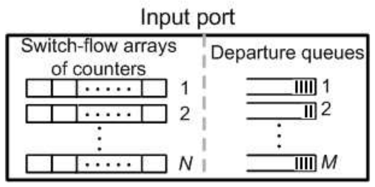
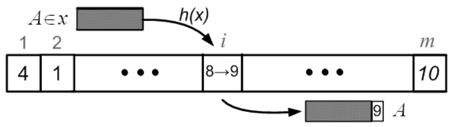

(c) Output resequencing structure.

Fig. 2. *Baseline* scheme


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


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

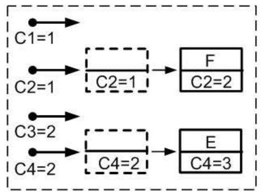
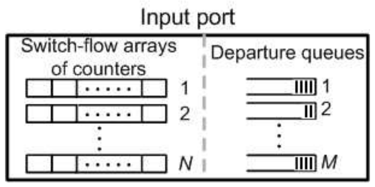
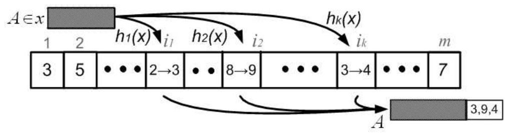

(c) Output resequencing structure.

Fig. 3. *Hashed-Counter* scheme


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


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

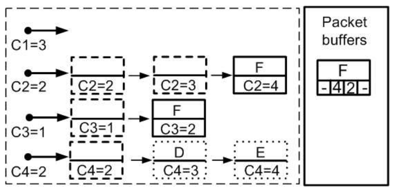

(c) Output resequencing structure.

Fig. 4. *Multiple-Counter* scheme

B. Output Resequencing Buffer

As shown in Figure 4(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.

In this example, we use $k = 2$. 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. However, in the list of $C4$, D and E are temporally kept, however, they are marked. When the missing packet B in the list departures, 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 before for the missing packets of counter. In addition, the real packet F is inserted in the packet buffer.

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

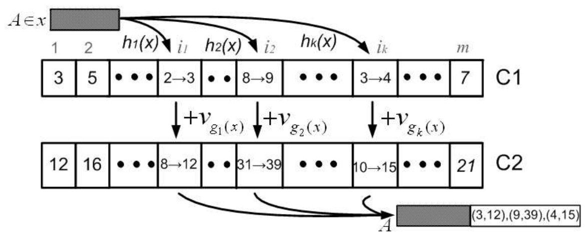


Fig. 5. B_h *Multiple-Counter* scheme in the input, packet A from flow x.

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 5. The scheme is based on B_h sequences [23].

Definition 1 (B_h Sequence): Let $(\mathcal{A}, +)$ be an abelian (commutative) group. Let $D = \{v_1, v_2, ..., v_\ell\} \subseteq \mathcal{A}$ be a sequence of elements of $\mathcal{A}$. Then D is a B_h *sequence* over $\mathcal{A}$ 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. 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 $\mathcal{A} = \mathbb{Z}$ and $D = \{v_1, v_2, v_3, v_4\} = \{1, 2, 5, 7\} \subseteq \mathcal{A}$.

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 5, the scheme uses two sets of k hash functions based on the flow ID: $\{h_1, \ldots, h_k\}$, with range $\{1, \ldots, m\}$, and $\{g_1, \ldots, g_k\}$, with range $\{1, \ldots, \ell\}$. 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.

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 same difference for the second counter minus $v_{g_i(P)}$.

As in the *Multiple-Counter* scheme, if for at least array entry $d_1(i) = 0$, i.e. the first counter of the packet is the next expected one, 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 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 $v_{g_i(P)}$.

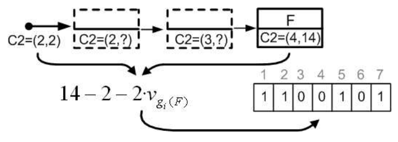


Fig. 6. Output resequencing structure for B_h *Multiple-Counter* scheme.

We now consider again the previously suggested example 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 counter value 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 $C2$. Due to the missing packets $B \in x, C \in y$ (with the variable increment 2 and 7), we have that $d_1(2) = 2$ and $d_2(2) = 2+7+5-5 = 14-5 = 9$. 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. Similarly, we can release F based on $C3$.

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 6, 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 one element equals of D and therefore F can be released.

V. Performance Trade-Off Overview

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

TABLE I
SCHEME COMPARISON

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

It first shows the *collision probability*, i.e. the probability that two arbitrary packets of a given switch flow use the exact same counters. 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 $\binom{m}{k}^{-1}$. Thus, the collision probability is significantly smaller. For the B_h *Multiple-Counter* scheme, collision happens when all the k choices form the B_h sequencing are the same. Consequently, the collision probability is further reduced to $\binom{m}{k}^{-1} \ell^{-k}$.

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 overheard. For instance, for $k = 2$, a cell size of 128B and a counter size of 2B. The *Multiple-Counter* scheme adds $4 - 2 = 2\text{B} = 1.6\%$ overheard, and the B_h *Multiple-Counter* scheme adds $8 - 2 = 6\text{B} = 4.7\%$ overheard.

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 also needs 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.

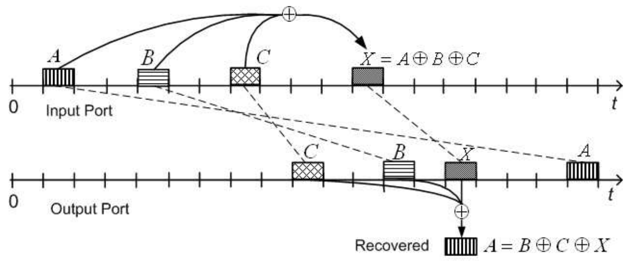


Fig. 7. Network coding example

VI. Buffer Cost Optimization

In each of the three suggested schemes, the packet resequencing delay is a function of the number of counters m. As less time a packet stays in a resequencing buffer, smaller resequencing buffers are required. However, more counters means more cost for the memory size in the input port, as indicates in second column of Table I. Due to this trade-off, we tend to properly choose the number of counters in each scheme to minimize the overall memory cost.

Consider a specific pair of input and output, its cost consists of two parts: the cost of the input memory size and the resequencing buffer in the output. Hence, it is a function of m and can be measured as

$$C(m) = \alpha \cdot m + \beta \cdot L \cdot B(m), \tag{1}$$

where $B(m)$ is the average resequencing buffer size, L is the packet size in bytes, and α, β are the memory cost coefficients per an input memory counter and a resequencing buffer byte, respectively. The total cost for all the input and output pairs is of course $N^2 \cdot C(m)$. $B(m)$ can be calculated by *Little's Law* as

$$B(m) = \lambda \cdot W(m), \tag{2}$$

where λ is the packet arrival rate for each input and output pair and $W(m)$ is the average time a packet stays in the resequencing buffer. Given the distribution of the total delay of a packet $T_T(m)$ (e.g., Equations (12), (13)), and the queueing delay T_Q, $W(t)$ can be presented as $E(T_T(m)) - E(T_Q)$, where $E()$ is the expected value function of a random variable. Finally, we rewrite Equation (1) as

$$C(m) = \alpha \cdot m + \beta \cdot L \cdot \lambda \cdot (E(T_T(m)) - E(T_Q)). \tag{3}$$

We can compare the first order derivative of $C(m)$ to zero to get the optimal value of m.

VII. Network Coding

A. Coding Against Rare Events

While the counter schemes above can reduce reordering delay, the total packet delay is necessarily lower-bounded by the *worst-case* queueing delay. For instance, 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 [24], [25]. 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, we send a redundant protection packet that contains this XOR of the previous packets, and re-initialize the XOR computation.

Figure 7 illustrates an example of use of the protection XOR packet. In the input port, the XOR packet X is generated using 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

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

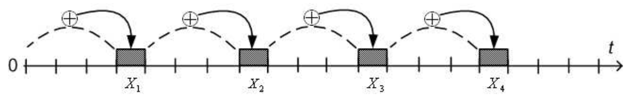
As we can see, packets A, B and C can depart the switch *before* the original packet A even arrives at the output. Thus, in this example, the XOR packet mainly helps reducing A's queueing delay — and we do not really care about its impact on channel capacity, as in typical network-coding examples.

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.

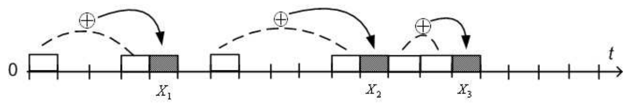
B. Network Coding Schemes

There are several possible network coding schemes to decide when the protection XOR packets should be sent. First, as shown in Figure 8(b), a simple coding scheme is to generate the XOR packets every L slots by taking the XOR of the last L packets, where $L = 3$ in the figure. The XOR packet is then inserted following the last of the L slots, and covers the L slots. These $L + 1$ resulting slots make up 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. The scheme overhead is clearly $\frac{L+1}{L} = 1 + \frac{1}{L}$.

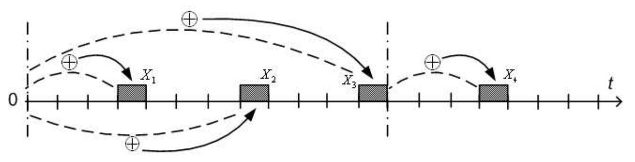
The scheme is simple to analyze, and we will model it in the remainder of the paper. However, it can practically be further improved in two directions. First, if the traffic load of the flow is light, the number of packets in a frame could be low, and the frame could even be empty. Therefore, the contribution of XOR packets does not justify the high relative overhead and



(a) Coding every 3 slots



(b) Coding every 2 packets



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

Fig. 8. Network coding schemes

additional delay caused by these XORs. It would make more sense to insert protection XOR packets every H packets instead of every L slots. Figure 8(a) illustrates such a scheme with $H = 2$, where the blank boxes stands for the regular packets and dark boxes for protection XOR packets.

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 L slots, the protection XOR packet cumulatively covers not only the last L 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 8(c) illustrates this cumulative coding scheme with a mega-frame consisting of three frames, each frame having $L = 3$ 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.

C. Space Coding

Like all the schemes proposed in this paper, network coding can also be applied in more complex switch architectures, with more than the three stages illustrated in Figure 1. For instance, it could apply to multi-stage switches with five stages [9]. It could likewise apply to fat-tree data center topologies with seven stages [10]–[12]. In all these other architectures, the proposed schemes would essentially remain unchanged.

However, there is an interesting improvement to make in architectures in which the second-stage consists of *multiplexing switches* that get all the traffic sent by several first-stage inputs. In particular, the data center topologies displayed in [10]–[12] satisfy this property, with the multiplexing switches alternatively called ToR (Top-of-Rack) and Edge Switches.

In such architectures, the multiplexing switches could take the XOR of all the packets sent by the diverse inputs to the same output, thereby coding over *space* in addition to coding over time. In other words, this coding would treat all flows from all inputs towards a given output as a single flow. However, the output would then need to use the information provided by the XOR packet across more than one reordering-buffer linked list, and therefore cannot see all these flows as a single flow. Therefore, such a scheme is expected to slightly increase the decoding complexity, yet decrease the expected reordering delay.

Such network coding can be generalized to any switch that receives packets from different inputs towards a given output, such that the packets are to take again multiple paths afterwards — for instance, the third stage of a seven-stage data center topology. However, since coding would be realized on packets coming from different inputs with previous variable queueing delays, such coding would need to include their IDs, which would greatly reduce the appeal of this generalized scheme.

VIII. Fair Reordering

A. Unfairness in Reordering Delay

In the switch, *the heavy and bursty flows will tend to suffer more from reordering delays*. This is because two packets in the same flow are reordered when the difference between their queueing times exceeds their inter-arrival time — and heavy and bursty flows have a higher probability of small inter-arrival times.

Therefore, there is an inherent *unfairness in reordering delay* among flows. The heavy and bursty flows (called *elephants* for simplicity) will experience higher delays than smaller and smoother flows (called *mice*).

This unfairness, apparently unnoticed previously, can have two detrimental consequences. First, a scheme that keeps flow order would typically cause higher delays for elephants, even those that are delay-sensitive. Even worse, in the *Baseline* scheme that preserves switch-flow order, elephant flows can also significantly impact mice that happen to share the same input and output.

Therefore, our goal will be to provide schemes that can decrease the unfairness between elephants and mice and make all packets experience similar reordering delay. In all schemes, we will assume that there already exists a mechanism to distinguish between elephants and mice at the input ports, and focus on providing fairness. We will show how we trade off the delay of elephants against the delay of mice using three schemes: priority queueing, variable numbers of counters, and variable network coding.

Incidentally, note that we might question why we would even want to *not* favor mice. After all, mice are typically seen as delay-sensitive UDP flows, while elephants would be delay-insensitive TCP flows. In fact, in a typical switch implementation, delay-sensitive flows would be given the same higher strict priority, and delay-insensitive flows would be assigned a lower priority. Therefore, our mechanisms would typically aim at reducing the unfairness between delay-sensitive flows — say, videoconferencing elephant flows vs. simpler VoIP mice.

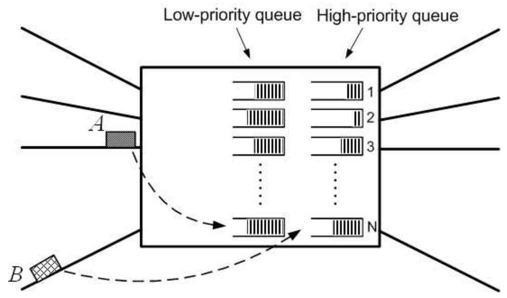


Fig. 9. Priority scheme implementation

B. Priority Queueing

In this scheme, we simply change the switch architecture to provide different priorities for elephants and mice. Elephants are assigned a higher priority, while mice get lower priority. We then provide more scheduling opportunities to elephants than to mice. Therefore, *we make the queueing delay unfair in order to make the total delay fair*.

Consider Figure 9, which illustrates a middle element, as part of the general switch architecture previously shown in Figure 1. The middle element includes $2N$ queues instead of N queues, i.e. high-priority elephant queues and low-priority mice queues.For instance, packet A belongs to a low-priority flow, while packet B belongs to a high-priority flow. They are both heading to the Nth output port. Then packet B is inserted into the high-priority queue, while A joins the low-priority queue.

The schedule of high-priority and low-priority queues would rely on Weighted Fair Queueing, providing a higher weight to the high-priority queue. It could then dynamically change the weights depending on the total delay experienced by the two types of traffic, as provided in feedback from the output ports. Therefore, this scheme could readily converge to a fair equilibrium, albeit at the cost of architectural changes and feedback-based fairness mechanisms.

C. Variable Number of Counters

The *Multiple-Counter* scheme can also be altered to become more fair. To do so, *flows with higher priority can be assigned more counters*. Then, the probability that high-priority packets will experience collision on all of their counters would decrease, because they have more counter. On the contrary, lower-priority packets would actually experience more collisions, because of the increased number of counters used by the high-priority packets.

Figure 10 illustrates an example for the counter assignment of two flows: flow x with high priority and flow y with low

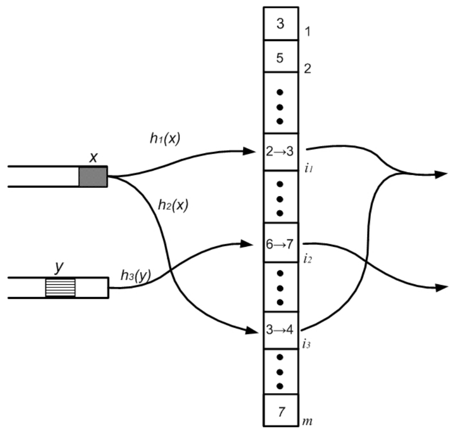


Fig. 10. Variable number of counters.

priority. The packets of flow x are assigned counters by the *Multiple-Counter* scheme with $k = 2$, while packets of flow y only use $k = 1$ counter, as in the simpler *Hashed-Counter* scheme. In this way, the variable number of counters enables the switch to get closer to a fair result by altering the resulting reordering delay, which was expected to be higher for flow x if the number of counters were equal for x and y. However, note that this scheme does not enable a smooth continuous transition to fairness, since the number of counters is both discrete and hard to change in the middle of the switch operation without incurring a higher complexity on the reordering buffer management scheme.

D. Variable Network Coding

In the same way as we provided more counters to high-priority flows, we could also envision *providing more protection packets to these high-priority flows*. For instance, we could distinguish high-priority and low-priority flows at the inputs, and provide schemes with different frame sizes for each type of flow. Even better, we could decide to only provide protection packets to bursts of a given size. In this way, we do not need to categorize flows, and can reserve protection packets to temporary bursts that are expected to generate a high reordering delay.

IX. 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. $F_Q(i) = \Pr(T_Q \leq i) = 1 - (1 - \rho)^{(i+1)}$. 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. Due to space constrains the model of the performance of the B_h *Multiple-Counter* scheme is not brought here.

B. Delay Distributions

We now want to analyze the *Baseline*, *Hashed-Counter*, and *Multiple-Counter* schemes. Note that the performances of the *Hashed-Counter* scheme and *Multiple-Counter* scheme 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. Its results are brought with brief proofs right after the following three theorems.

The first theorem is about the Rare-Event delay model. Note that 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 $i \in [0, T - 1]$,

$$\Pr(T_T \leq i) = (1 - \epsilon) \cdot (1 - p\epsilon)^{(T-i-1)}. \tag{4}$$

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

$$\Pr(T_T \leq i) = (1 - \epsilon) \cdot (1 - \frac{p}{m} \cdot \epsilon)^{(T-i-1)}. \tag{5}$$

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

$$\Pr(T_T \leq i) = (1 - \epsilon) \cdot \left(1 - (1 - p_k^{T-i-1})^k\right) \tag{6}$$

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

$$\Pr(T_T \leq i) = (1 - \epsilon) \cdot \left(1 - (1 - p_{h,k}^{T-i-1})^k\right) \tag{7}$$

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

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. ■

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,

$$\Pr(T_T \le i) = F_Q(i) \cdot \prod_{j=1}^{\infty}(1-p\cdot(1-F_Q(i+j))). \qquad (8)$$

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

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

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

$$\Pr(T_T \le i) = F_Q(i) \cdot (1-(1-p_{k,i})^k). \qquad (10)$$

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

Proof: (i) In the General delay model, a packet arrived 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. ■

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

$$\Pr(T_T \le i) = (1-(1-\rho)^{(i+1)}) \cdot \prod_{j=1}^{\infty}\left(1-p\cdot(1-\rho)^{(i+j+1)}\right). \quad (11)$$

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

$$\Pr(T_T \le i) = (1-(1-\rho)^{(i+1)}) \prod_{j=1}^{\infty}\left(1-\frac{p}{m}\cdot(1-\rho)^{(i+j+1)}\right). \quad (12)$$

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

$$\Pr(T_T \le i) = (1-(1-\rho)^{(i+1)}) \cdot (1-(1-p_{k,i})^k). \qquad (13)$$

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

We now want to generalize the last results to a more general model in which the flow sizes are not necessarily negligible, and as a consequence the probability that two different packets share the same flow has to be considered. For simplicity, we assume that in the recent time slots the probability for an arrival of a packet of the same flow as the current packet is independent in the time difference of these two packets. We denote this probability by p_s. Since the total arrival rate 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 $p_d=p-p_s$. We again consider a delayed packet that enters its input port in time $t=t_0$.

With these assumptions, we now present a new version of Theorems 1, 2.

We start again with the Rare-Event delay model.

Theorem 4 (Extended Rare-Event delay model): (i) Using the *Baseline* scheme, for $i\in[0,T-1]$,

$$\Pr(T_T \le i) = (1-\epsilon)\cdot(1-p\epsilon)^{(T-i-1)}. \qquad (14)$$

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

$$\Pr(T_T \le i) = (1-\epsilon)\cdot(1-(p_s+\frac{p_d}{m})\cdot\epsilon)^{(T-i-1)}. \qquad (15)$$

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

$$\Pr(T_T \le i) = (1-\epsilon)\cdot\left(1-(1-p_k^{T-i-1})^k\right) \qquad (16)$$

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

$$\Pr(T_T \le i) = (1-\epsilon)\cdot\left(1-(1-p_{h,k}^{T-i-1})^k\right) \qquad (17)$$

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

Proof: (i) The probability is the same as in the original model since in this scheme, we cannot distinguish between packets of different flows that share the same pair of input and output nodes.
(ii) The total delay is at most i, if the packet is not delayed (w.p. $(1-\epsilon)$) and its counter was not used by a delayed packet in $(T-i-1)$ time slots. Since a packet of the same flow must use the same counter while a packet of a different flow use this counter w.p. $\frac{1}{m}$, the results follows.
(iii) In the *Multiple-Counter* scheme, a packet can leave the switch if *at least one* of its counters k is in order. In a given time slot, a counter is used by a delayed packet w.p. $p_k=1-(p_s+p_d\cdot\frac{k}{m})\cdot\epsilon$. It is used by a delayed packet at least once in $T-i-1$ time slots w.p. $(1-p_k^{T-i-1})$. Therefore, the probability of at least one counter is in order counter is $\left(1-(1-p_k^{T-i-1})^k\right)$.
(iv) We remind that in the B_h *Multiple-Counter* scheme, a packet can leave the switch if in *at least one* of its counters

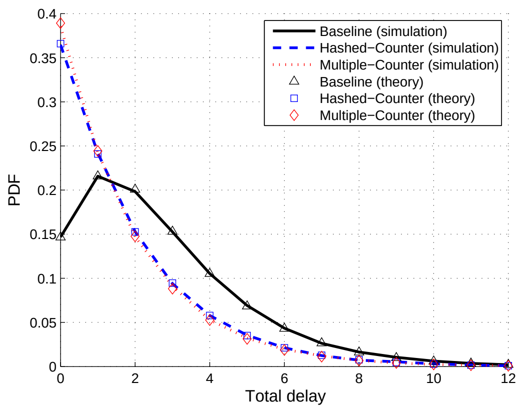


Fig. 11. PDF of the total delay for the geometric delay model.

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. In each time slot, each of the k counters is not used by a delayed packet w.p. $(1 - p_d \epsilon \frac{k}{m} - p_s \epsilon)$. If this counter is used, the specific value of D is not used w.p. $\frac{\ell-1}{\ell}$ only if it is a packet of a different flow. ■

We continue to the General delay model.

Theorem 5 (Extended General delay model): (i) Using the *Baseline* scheme,

$$\Pr(T_T \leq i) = F_Q(i) \cdot \prod_{j=1}^{\infty} (1 - p \cdot (1 - F_Q(i + j))). \tag{18}$$

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

$$\Pr(T_T \leq i) = F_Q(i) \cdot \prod_{j=1}^{\infty} \left(1 - (p_s + \frac{p_d}{m}) \cdot (1 - F_Q(i + j))\right). \tag{19}$$

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

$$\Pr(T_T \leq i) = F_Q(i) \cdot (1 - (1 - p_{k,i})^k). \tag{20}$$

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

Proof: (i) Again, the probability is not changed since in this scheme we do not distinguish between packets of different flows.
(ii) Likewise, the probability that the current counter is used by a different packet is $(p_s + \frac{p_d}{m})$.
(iii) A packet uses the same counter and is delayed for at least $(i + j)$ time slots w.p. $(p_s + p_d \cdot \frac{k}{m}) \cdot (1 - F_Q(i + j))$ ■

X. Simulations

A simulator is developed for various use scenarios, including geometric delay model, Clos switch and data center switch. To better mimic the practical network behavior, we further implement the feature to feed the flow source with real-life traces [26]. Simulations for each scenario are conducted as described below.

A. Geometric Delay Model Simulations

We run simulations for the geometric delay model discussed in Section IX-B. In the simulation, the number of flows equals 2^{27}, so that the assumption that all flows are mice of negligible size still holds. We generate the switch flow using Bernoulli i.i.d traffic with arrival rate $p = 0.98$, the parameter of geometric delay $\rho = 0.4$, 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 11 depicts the simulation results, compared with the theoretical results from (11), (12) and (13). The simulation results match the theoretical results quite well. Furthermore, we can see that changing the definition of ordering from *switch flow ordering* to *flow ordering* significantly decreases the average and standard-deviation of both delays. Using the *Hashed-Counter* scheme, the average total delay is drastically reduced from 2.67 to 1.64 time slots. The improvement from the *Hashed-Counter* scheme to the *Multiple-Counter* scheme is not significant for the total delay. The reason is that here the queueing delay and the internal resequencing delay contribute most to the total delay, which are of the part that the *Multiple-Counter* scheme cannot help. However, more noticeable improvements can be observed in other network scenarios, which will be shown later.

Figure 12(a) shows the average resequencing delay as a function of the number of total counters. The *Hashed-Counter* and *Multiple-Counter* scheme significantly reduce the resequencing delay. When $m \geq 20$, the *Multiple-Counter* scheme makes the resequencing delay very close to 0.

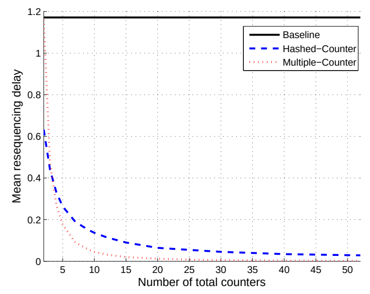
B. Switch Simulations

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.

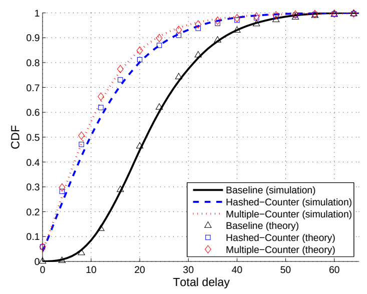
We start by comparing the performance of hash-based counter schemes by using 4096 flows per (input, output) pair, i.e. a total of $4096 \cdot 16 = 65,536$ flows. The total load is set to $p = 0.95$. Each flow is generated using Bernoulli i.i.d. traffic of parameter $\rho/65,536$. In the *Multiple-Counter* and B_h *Multiple-Counter* schemes, we have $k = 2$.

Figure 12(b) compares the results of the switch simulations to the theoretical results from (11), (12) and (13), similarly to Figure 11. 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 for the switch simulations the assumption that the flows are mice of negligible size does not hold.

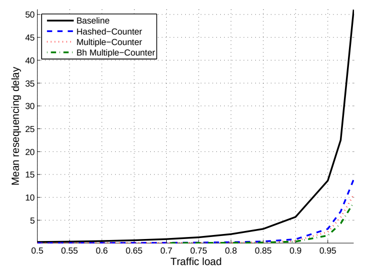
Figure 12(c) plots the average resequencing delay in the switch as a function of the traffic load ρ. As the load increases



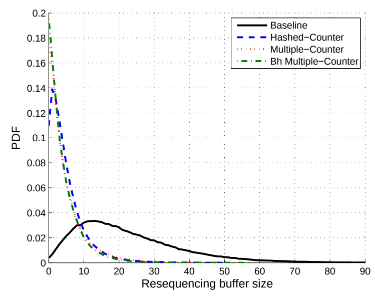
(a) Mean resequencing delay for the geometric delay model as a function of the number of total counters.



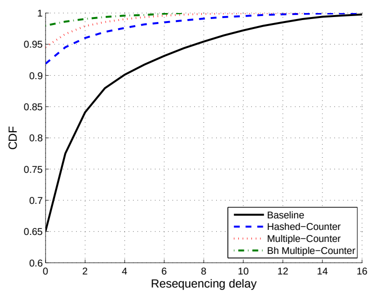
(b) Comparison of the switch to the geometric delay model.



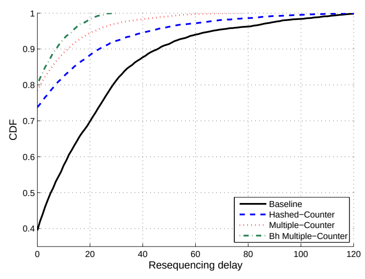
(c) Mean resequencing delay for the switch as a function of the traffic load.



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



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



(f) CDF of the resequencing delay for a data center, using a real-life trace.

Fig. 12. Simulation results

and the delays become large, 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 a lower 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 16%.

Figure 12(d) shows the PDF 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 *Hashed-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 [26]. We use a real hash function [27] 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, Figures 12(e) show 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. Data Center Simulations Using Real-Life Traces

Simulations are conducted with a 5-stage data center structure described in [12]. The input ports are fed with real-life traces as Section X-C. The total number of counters in the *Hashed-Counter* scheme and *Multiple-Counter* scheme is $m = 20$. For the B_h *Multiple-Counter* scheme, $m = 10$. The *Multiple-Counter* scheme and B_h *Multiple-Counter* scheme set the multiple counters as $k = 2$.

Figure 12(f) 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. More encouragingly, the improvement from the *Hashed-Counter* scheme to the *Multiple-Counter* scheme is much more satisfactory. 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 reordering caused by other flows becomes more severe. This can well be improved by the *Multiple-Counter* scheme. However, the best results are obtained by using the B_h *Multiple-Counter* scheme, for which the resequencing delay is only 1.84 time slots.

XI. Conclusion

In this paper, we provided schemes to deal with packet reordering, an emerging key problem in next-generation switch designs. We first 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 10.

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.

Acknowledgement

The work was partly supported by the European Research Council Starting Grant n°210389. The authors would like to thank N. McKeown and G. Varghese for suggesting the problem and solution presented in Section I-E.

References

[1] "Cisco CRS-1 multishelf system." [Online]. Available: http://www.cisco.com/en/US/products/ps5842/

[2] "Juniper TX matrix plus." [Online]. Available: http://www.juniper.net/us/en/products-services/routing/t-tx-series/txmatrix-plus/

[3] "Dune-Broadcom FE600 fabric element." [Online]. Available: {http://www.dunenetworks.com/webSite/Modules/News/NewsItem.aspx?pid=355\&id=89}

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

[5] "Latency and jitter: Cut-through design pays off for arista, blade." [Online]. Available: http://news.idg.no/cw/art.cfm?id=4033D2EF-1A64-6A71-CE43F39B8EDA3B3A

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

[7] 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.

[8] 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.

[9] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, and N. McKeown, "Scaling internet routers using optics," *ACM SIGCOMM*, vol. 33, no. 4, pp. 189–200, 2003.

[10] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *SIGCOMM*, 2009, pp. 51–62.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008, pp. 63–74.

[12] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *SIGCOMM*, 2009, pp. 39–50.

[13] J. S. Turner, "Resequencing cells in an ATM switch," Washington University, Computer Science department, WUCS-91-21, 2/91.

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

[15] J. S. Turner, "Resilient cell resequencing in terabit routers," in *Washington University, Computer Science department*. [Online]. Available: www.cse.seas.wustl.edu/techreportfiles/getreport.asp?285

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

[17] J. J. Jaramillo, F. Milan, and R. Srikant, "Padded frames: a novel algorithm for stable scheduling in load-balanced switches," *IEEE/ACM Trans. Networking*, vol. 16, no. 5, pp. 1212–1225, 2008.

[18] B. Wu, Y. Xu, H. Lu, and B. Liu, "A practical packet reordering mechanism with flow granularity for parallelism exploiting in network processors," in *IPDPS*, 2005.

[19] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "A hardware packet resequencer unit for network processors," in *Architecture of Computing Systems (ARCS)*, no. 4934, 2008, pp. 85–97.

[20] Y. Nebat and M. Sidi, "Parallel downloads for streaming applications - a resequencing analysis," *Perform. Eval.*, vol. 63, no. 1, pp. 15–35, 2006.

[21] W. Xiao and D. Starobinski, "Extreme value fec for wireless data broadcasting," in *IEEE Infocom*, 2009.

[22] Y. Xia and D. N. C. Tse, "Analysis on packet resequencing for reliable network protocols," *Perform. Eval.*, vol. 61, no. 4, pp. 299–328, 2005.

[23] S. Graham, "B$_h$ sequences," *Analytic Number Theory*, vol. 1 (Allerton Park, IL, 1995), 1996.

[24] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. K. Ramakrishnan, "LT-TCP: End-to-end framework to improve TCP performance over networks with lossy channels," in *IWQoS*, 2005, pp. 81–93.

[25] V. Sharma, S. Kalyanaraman, K. Kar, K. K. Ramakrishnan, and V. Subramanian, "MPLOT: A transport protocol exploiting multipath diversity using erasure codes," in *INFOCOM*, 2008, pp. 121–125.

[26] C. Shannon, E. Aben, K. Claffy, and D. E. Andersen, "CAIDA Anonymized 2008 Internet Trace," http://imdc.datcat.org/collection/.

[27] "Integer hash function." [Online]. Available: http://www.concentric.net/~Ttwang/tech/inthash.htm