Scaling Multi-Core Network Processors Without the Reordering Bottleneck

Alexander Shpiner*, Isaac Keslassy* and Rami Cohen§
*Technion, {shalex@tx, isaac@ee}.technion.ac.il
§IBM Research, ramic@il.ibm.com

***Abstract*—Today, designers of network processors strive to keep the packet reception and transmission orders identical, and therefore avoid any possible out-of-order transmission. However, the development of new features in advanced network processors has resulted in increasingly parallel architectures and increasingly heterogeneous packet processing times, leading to large reordering delays.**

In this paper, we introduce novel scalable scheduling algorithms for preserving flow order in parallel multi-core network processors. We show how these algorithms can reduce reordering delay while adapting to any load-balancing algorithm and keeping a low implementation complexity overhead. To do so, we use the observation that all packets in a given flow have similar processing requirements and can be described with a constant number of logical processing phases. We further define three possible knowledge frameworks of the time when a network processor learns about these logical phases, and deduce appropriate algorithms for each of these frameworks. Finally, we model our proposed algorithms and simulate them under both synthetic traffic and real-life traces, and show that they significantly outperform past approaches.

I. Introduction

A. Background

Network Processors (NPs) are specialized software-programmable architectures in routers, switches and network cards. NPs are designed to process packets at high speeds, and especially to implement such diverse functions as forwarding, classification, protocol conversion, DPI, SSL and firewalling [1]–[4].

NPs are required to avoid out-of-order transmission of the packets, because out-of-order packets can disrupt the local pipeline logic of the router, as well as significantly decrease the throughput of TCP flows [5], [6]. Unfortunately, in recent years, two trends have made it increasingly hard for NP designers to keep packets in order without suffering from a large additional delay. First, NP architectures are becoming *increasingly parallel*. For instance, they often rely on many parallel processing cores (e.g., the Cavium CN68XX [7] or the AMCC nP7310 [8]), or on a hybrid combination of parallel and pipeline cores (e.g., the EZChip NP-4 [9] or the Netronome NFP-32xx [10]). Second, packet processing needs are becoming *increasingly heterogeneous*. NPs need to implement a growing number of increasingly complex features, such as advanced VPN encryption, LZS decompression, VoIP SBC, video CAC, per-subscriber queueing, and hierarchical classification for QoS [7], [11], [12].

As a consequence of these two trends of parallel architectures and heterogeneous processing delays, many packets with small processing times may be ready to leave the NP, but need to wait for a few packets that arrived earlier and are still stuck in heavy processing tasks. Therefore, these NPs exhibit high and unpredictable *reordering delays*, which conflict with the delay requirements that NP designers need to meet.

Current reordering algorithms typically do not handle this heterogeneous traffic gracefully. In particular, as detailed below, they either (a) cause a needless large reordering delay, or (b) rely on a static load-balancing through hashing and can cause a lower throughput.

In this paper, our goal is to provide a scalable algorithm that reduces the reordering delay, while adapting to any load-balancing scheme. Thus, the NP designer can keep using the same load-balancing scheme that achieves high throughput, and just apply our algorithm to reduce the reordering delay.

As illustrated in Figure 1, consider a network processor with N processing elements (PEs). Assume that a packet A arrives before a packet B, and both are sent to different PEs. Further assume that A and B belong to different flows, and that B has light processing requirements and is ready to leave quickly, while A has heavy processing requirements. For instance, B only needs forwarding, while A also needs DPI.

Today, B typically needs to wait for A, potentially incurring a high delay. In this paper, we suggest to distinguish packets based on their logical processing requirements. We use the observation that packets belonging to a single flow can be defined as having similar logical processing requirements. Therefore, if A and B have different logical processing requirements, they belong to different flows, and B does not need to wait for A in order to preserve flow order. Thus, by defining ordering domains based on logical processing requirements, we can still preserve flow order while significantly decreasing reordering delay.

B. Our Contributions

In this paper, we present a new packet reordering algorithm for multi-core parallel NPs. Our reordering algorithm achieves a low reordering delay, while working with *any* load-balancing scheme, thus also obtaining a maximal throughput.

We provide *three core contributions* in this paper. First and foremost, we introduce a new model for NPs. In particular, we make the observation that all the packets of a given flow can typically be divided into an equal number of well-defined logical processing phases, which correspond to their similar processing

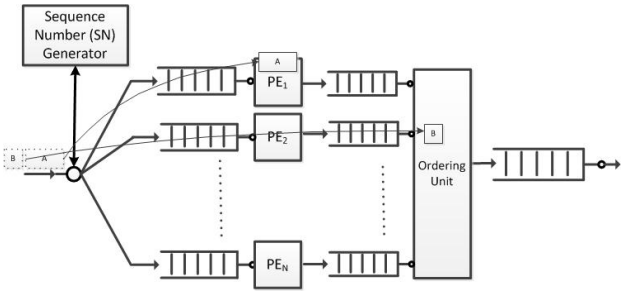


Fig. 1. Reordering example in an NP architecture. Packet B arrives after packet A, but has smaller processing needs and finishes its processing earlier. If packets belong to different flows, waiting for A would lead to an unnecessary reordering delay. In this paper, we suggest to redefine logical processing requirements, such that if A and B have different logical processing requirements, then A and B necessarily belong to different flows, and B can leave while keeping flow order.

requirements.

Then, we also introduce three knowledge frameworks that define the stages at which the NP learns about the number of processing phases: either (1) as packets arrive, or (2) as they start being processed, or (3) as they complete processing. These stages depend on the NP design assumptions.

Next, we make an algorithmic contribution. Based on each of these knowledge frameworks we introduce three algorithms, called Reordering Per Processing Phase (RP^3), which leverage this knowledge to reduce the reordering delay. We further illustrate how these algorithms are implemented, with an increasingly complex implementation as this knowledge is reduced.

Last, in a more theoretical contribution, we develop mathematical models that compare the total packet delay under different reordering algorithms and different knowledge frameworks. These models provide some intuition on why our suggested RP^3 algorithms are more efficient.

Finally, using extensive simulations based on both synthetic traffic and real-life traces, we analyze our RP^3 algorithms and show that their reordering delays are negligible when compared to previously known techniques. We also illustrate how a lower variability in the delays of the logical processing phases leads to significant improvements in the reordering delays of our algorithms.

C. Related Work

Recent research works have described several architectures that aim to reduce the reordering delay. First, pipeline-based architectures without parallelism clearly preserve the packet order. However, they are hardly scalable, because of the heterogenous requirements of the packets, the synchronization requirements, and the granularity of the processing commands [13]. Thus, we further discuss only parallel architectures.

Statically mapping each flow to a single core using hashing is another popular way to intrinsically avoid reordering [14], [15]. However, it results in an insufficient utilization of the cores, and therefore in a lower throughput, due to the fact that several *elephant* flows may map to the same core [16]. Moreover, it is possible to adapt the load-balancing scheme by using feedback on the core utilization in order to increase throughput [17], [18]. However, this can also cause packet reordering [19], and therefore requires an ordering mechanism. In addition, all these approaches fix the load-balancing scheme, while we would like to adapt to any potential scheme.

The multipass network processor is a recent solution that can reduce the reordering delay for short-processed packets by preempting the processing of heavy-processed packets [11], [20], [21]. However, the usage of such a capability results in possible starvation of packets with heavy processing requirements, and in a context switch loss, which can be associated with a significant overhead for storing and loading the packet state, thus reducing the overall system performance, and also impacting the software development flexibility. In addition, the multipass NP still requires a mechanism to preserve packet order.

There are several algorithms for keeping packet order without changing the load-balancing scheme. First, the NP can allocate a global sequence number to every arriving packet [22], [23]. This is illustrated in Figure 1, assuming that the sequence number follows a global counting. Then, the ordering unit is simple to implement based on this global numbering. However, as explained above, this solution also incurs a high reordering delay, because all packets are treated as a single flow, the order of which has to be preserved. A second, ideal approach is to rely on per-flow sequencing [23]–[26], thus providing a minimal reordering delay. However, as explained in [16], it is not scalable to a large number of network flows because of the large number of needed counters. A third method to avoid packet reordering is to keep an inter-thread signaling system between the cores [22]. However, in NPs with a high degree of parallelism and a high clock frequency, this method can be complex to implement.

A fourth and last appealing approach is to statically aggregate flows by hashing the flow identifier in the packet header into several ordering domains [16]. A different sequence number generator is then assigned for each ordering number. The flow identifier may for instance consist of its 5-tuple and its input interface. In each of the ordering domains, the order of the packets is preserved. However, an unnecessary reordering delay occurs between flows with different processing requirements that are hashed into the same ordering domain. In particular, this method suffers from the fact that flows in the same domain do not necessarily have similar processing delays. By contrast, we suggest to base reordering domains on the processing phases, thus following natural flow properties rather than using arbitrary hashing. Note that the two solutions can also be combined, by simultaneously distinguishing flows based on an arbitrary hash as well as on the number of processing phases.

More generally, the packet ordering problem in parallel systems is not new. It has been already discussed in the works on ATM switches. For example, [27] answered the challenge of limited sequence number field size in the packet header, and [28] answered the problem of correct ordering with packet loss possibility. However, those works did not try to reduce the reordering delay.

Finally, there are also research works on packet reordering in switches [29], [30]. However, those works assume equal processing times for all packets, and therefore their solutions

cannot apply to our case.

The rest of the work is organized as follows. We first define our system model in Section II. Then we provide insights on the impact of the phase processing delay variability in Section III. We further present our proposed algorithms in Sections IV and V, and discuss their possible implementation variations in Section VIII. The theoretical model of the reordering algorithms is provided in Section VI. Last, we present simulation results in Section VII, before concluding in Section IX.

II. System Model and Problem Formulation

A. System Model

Consider a network processor (NP) with N processing elements (PEs), which can also be general-purpose CPU cores. We allow for any load-balancing scheme. Therefore, each arriving packet can be allocated to any arbitrary PE for processing. As commonly assumed, we consider a single stream of incoming packets [16], [20], [24], [25], [31]. We allow arbitrary arrival times of the packets, and assume infinite buffer sizes. Also, we neglect the different priority levels of the flows. The management of the packets in the NP is performed with packet descriptors, which hold all the necessary information for packet processing, including pointers to the packets in the main buffer.

We mandate that *packets from the same flow depart in the order of their arrival*. When a packet needs to wait for another packet to make sure that it departs in order, it experiences a *reordering delay*. Our goal is to reduce the average reordering delay, using a simple and scalable algorithm.

B. Assumptions

The main assumption in this paper is that the processing requirements of all the packets from the same flow can be divided into *an equal number of logical processing periods*, denoted as *phases* (as in [20]). For instance, all the packets of a given flow may only require forwarding, while all the packets of another flow may need forwarding, deep packet inspection, and IPSec processing.

This requirement is reasonable in practice because phases are only logical, and we do not mandate that all phases correspond to the same number of cycles. For instance, if the first and last packets in a flow require a higher number of phases because the network processor needs to open or close flow-based states, then we can simply insert some empty dummy logical phases in the other packets. Of course, a higher variability of the actual time of each phase will adversely impact the reordering delay of any algorithm based on this assumption. We also allow large processing functions such as deep packet inspection to be further subdivided into different logical phases, as long as all the packets of a given flow experience the same subdivision. Finally, we denote by $\Phi \in \mathbb{N}$ the maximal possible number of processing phases per packet.

In practice, the required number of processing phases can be obtained after parsing the packet header, and comparing its content to the user-configured rules for specific flows or protocols. Therefore, the NP does not need to store the total number of phases for each of the flows. For example, if the packet includes an IPSec header, the configuration table may indicate that the packet needs to be authenticated and forwarded. Further, if an authentication needs ten processing phases and IP forwarding needs one processing phase, the NP deduces that the packet requires eleven processing phases.

In addition, if the processing cannot be subdivided into logical phases for some specific flows, then it is still possible to treat those flows as in current NPs. In particular, those flows can be sent to other ordering domains by hashing the packet header as in [16]. This type of ordering domain can coexist in parallel with our processing-based reordering domains.

C. Knowledge Frameworks

Estimating the processing time of each packet can significantly help in reducing the average reordering delay. Nevertheless, in many cases such an estimation can only be realized during, or at the end of the data processing.

In this section, we define *three independent knowledge frameworks* regarding the time at which the NP knows about the number of processing phases of each packet. In our discussions with industry NP designers, we found that these frameworks capture different assumptions and capabilities of NP vendors and architectures[1]. Our goal is to study the *impact* of each knowledge framework on the complexity of the reordering algorithm and on the resulting reordering delay. The following three frameworks are ordered from the framework with more knowledge to the framework with less knowledge, and therefore are expected to result in an increasingly high reordering delay. This is confirmed in simulations (Section VII).

The first framework assumes that there is a header parser before load-balancing, thus ensuring a *full knowledge* of the number of the phases of each packet before it is load-balanced among the PEs. This framework is the simplest one to analyze, and it conveys some intuition on the problems involved. It is often adopted in the literature [20], [31]–[35].

Framework 1: *The number of processing phases of a packet is known upon its arrival at the network processor.*

In many cases, once a packet arrives at a PE, its first processing phase includes packet classification, based on the packet headers (including in some cases the application header) [36]. This packet classification can help us in determining the number of remaining processing phases, leading to a second framework.

Framework 2: *The total number of processing phases of a packet is only known after its first processing phase.*

In most cases, we would expect the total number of phases to be known after the first processing phase. However, in cases where the processing of the packet is performed recursively without a predetermined number of iterations, as in MPLS or PBB label encapsulation [21], it is hard to know the number of processing phases in advance [37]. More generally, we can introduce a third framework, in which we only know the number of processing phases a packet has already gone through, but not the number of remaining ones.

[1]During our contacts with industry, we found that different NP companies made radically different assumptions about this knowledge. Therefore, we introduce different frameworks to fit different NP designers. Also, note that these different frameworks may entail different design costs.

Framework 3: *The PE only knows about the number of processing phases a packet has already gone through, but not about the number of remaining ones. Therefore, the total number of processing phases of a packet is known only after its processing is completed.*

III. Reordering Delay under Ideal and Real Settings

Our proposed architectures aim to reduce the average reordering delay by scheduling packets based on the current information regarding their number of required processing phases. As mentioned before, we assume that packet processing can be divided into well-defined phases, and that all the packets of a specific flow have the same number of phases.

In the paper, we allow the processing time of each phase to be variable. The processing time of a given phase can significantly vary in different runs, even for the same processing requirement and in the same processing unit. There are several reasons for such variability. On the one hand, using the cache for some data from an external resource can decrease the processing time of the specific code segment. On the other hand, access to a shared resource may extend the processing time, as the PE waits for its turn to use the shared resource.

However, the intuition in our proposed architectures is that *when the variability in the processing time of each phase is limited*, we can reduce the reordering delay and benefit more significantly from the knowledge on the number of phases of each packet. The next theorem justifies this intuition by showing thatif there were no phase processing delay variability, we could also potentially achieve zero reordering delay. Clearly, this assumption is only theoretical, and it is restricted to this theorem.

Theorem 1 (Ideal settings): Under Framework 1, assume that there is no phase processing delay variability, and therefore that all packets of a given flow experience the exact same processing delay. Further assume that there is perfect load-balancing between the PEs. Finally, assume that the set of all packets with a given delay is assigned its own ordering domain. Then no reordering delay will occur.

Proof: Assume by contradiction that reordering delay is not always zero. Consider then the first packet B that experiences a positive reordering delay, i.e. assume that there is another packet A such that A arrives before B at the NP while B later waits for A to depart the NP. Packets from different flows use different SN generators, and therefore do not cause reordering delay to each other. Thus, necessarily A and B have equal processing delays $d_A = d_B = d$, and use the SN generator for all packets of delay d.

In addition, assume that packet A arrived at the NP at time $t_{arr,A}$, and packet B at time $t_{arr,B} > t_{arr,A}$. The buffering delay of the packets is denoted as $\Delta_{buf,A}$ and $\Delta_{buf,B}$, respectively. Finally their transmission times (at which they depart the NP) are denoted as $t_{tr,A}$ and $t_{tr,B}$, respectively. The perfect load-balancing guarantees that the later packet B cannot bypass the earlier packet A in the queues, i.e.

$$t_{arr,A} + \Delta_{buf,A} < t_{arr,B} + \Delta_{buf,B}. \quad (1)$$

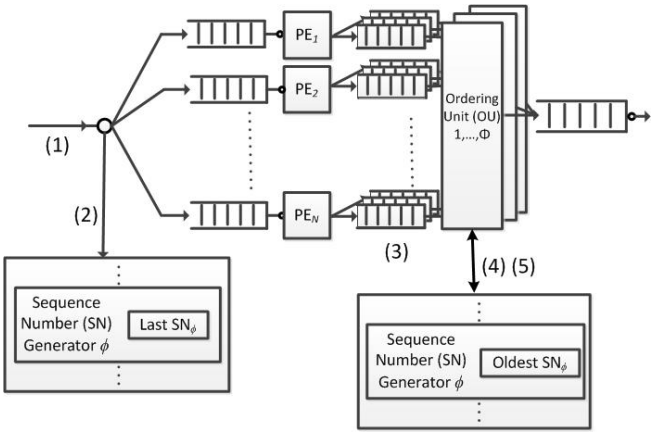


Fig. 2. RP^3 Algorithm under Framework 1. Packets with a different number of phases belong to different flows. Thus, they will not affect the reordering delay of each other.

By assumption, there is a constant processing time for each phase in the PEs, therefore the processing delay of both packets A and B is equal, hence

$$t_{arr,A} + \Delta_{buf,A} + d < t_{arr,B} + \Delta_{buf,B} + d \leq t_{tr,B}. \quad (2)$$

Since we assumed that packet A cannot experience a reordering delay, it means that necessarily

$$t_{tr,A} = t_{arr,A} + \Delta_{buf,A} + d,$$

i.e.

$$t_{tr,A} \leq t_{tr,B}.$$

Therefore, packet A will be transmitted before packet B, and no reordering delay will occur, hence contradiction. ■

IV. RP^3 Algorithm under Framework 1

In this section, we present our proposed order-preserving RP^3 (Reordering Per Processing Phase) algorithms that work under Frameworks 1 and 2. Since Framework 1 provides full knowledge of the number of phases, the resulting algorithm is relatively simple, and will provide some intuition on the mechanism before describing more complex algorithms under later frameworks with limited knowledge.

Intuitively, *the algorithm divides all the flows into subsets of flows that require the same number of processing phases*. The goal is to avoid a case in which a packet belonging to a flow with many processing phases blocks a packet belonging to a flow with few processing phases at the reordering unit, as illustrated in the Introduction. By reducing the amount of such blocking, the algorithm decreases the reordering delay.

A. RP^3 Algorithm under Framework 1

Figure 2 presents the RP^3 algorithm under Framework 1. The algorithm relies on an architecture that includes Φ sequence-number (SN) generators, and Φ ordering units (OUs). The ϕ-th SN generator assigns the next SN for packets with ϕ processing phases. The ϕ-th ordering unit tracks the latest released packet

with ϕ processing phases. Each ordering unit has N input buffers, one per PE. Note that while the ordering units are presented as separate for intuition, in practice they may be implemented on a single core.

First, each packet is immediately assigned an SN upon arrival, as shown in step (1) of Figure 2. In Framework 1, the number of required processing phases ϕ of the packet is known as soon as it arrives. Therefore, the SN of the packet is assigned by the ϕ-th SN generator (step (2) in Figure 2), which also increments its last assigned SN.

Next, the packet descriptor is sent by the load balancer to some PE i. After finishing the processing in PE i, the packet descriptor is placed in buffer i of ordering unit ϕ (step (3)). The ordering unit can only release in-order packets. To do so, each ordering unit ϕ checks if one of the N head-of-line packets in its buffers has an SN equal to the next expected SN, i.e. to the SN of the oldest packet in the NP with ϕ phases (step (4)). If the condition is met, the packet can depart, and the expected SN is incremented (step (5)). Else, the packets keep waiting for the next expected packet.

Each ordering unit ϕ preserves the order of the packets that require ϕ processing phases. Thus, *reordering delay can only occur within those packets*. For instance, assume that packets A and B belong to two different flows with the same number of logical processing phases, and A arrives to the NP before B, but B has completed its processing earlier. Then B will wait for A in order to depart, even though they belong to different flows, because they have the same number of phases. As we will show in simulations (Section VII), this reordering algorithm achieves a significantly low reordering delay, and this is especially true when the variability in the delay of each processing phase is low.

B. RP^3 Algorithm under Framework 2

Under Framework 2 the required number of processing phases of a packet is unknown upon arrival. It only gets known as it arrives to the PE and begins getting processed. Therefore, under Framework 2, the algorithm for Framework 1 cannot be directly used, since a phi-based SN cannot be assigned upon arrival, and the algorithm needs to be adapted.

Figure 3 illustrates the RP^3 algorithm under Framework 2. The algorithm relies on an architecture that includes a single sequence number (SN) generator, an array of $\Phi + 1$ linked lists, and Φ ordering units (OUs). Each ordering unit has N input buffers, one per PE. Each linked list observes an order of packets with the same number of processing phases. An additional linked list is added for the packet SNs with an unknown number of processing phases. A packet is assigned an SN upon arrival (step 1 in Figure 3). Its number of required processing phases ϕ is still unknown, according to Framework 2. Thus, the SN is added to the tail of the list of packets with an unknown number of phases (step 2 in Figure 3). The packet descriptor is then queued by a load balancer in the buffer of one of the PEs. When the packet arrives to its PE, its exact total number of processing phases ϕ is known after the first processing phase. Then, its SN is removed from the list of packets with an unknown number of phases and added to list ϕ (step 3 in Figure 3). After finishing the processing in PE i, the packet descriptor is placed in buffer i of the ordering

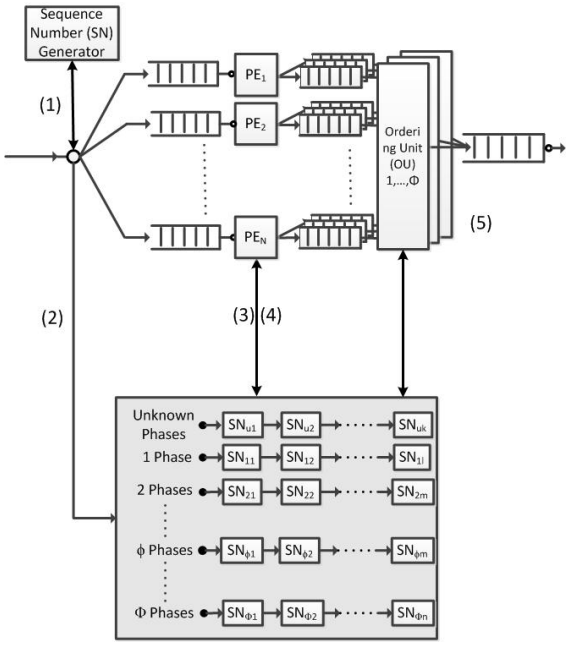


Fig. 3. RP^3 Algorithm under Framework 2. Each packet arrives with an initially-unknown amount of required processing. Therefore, it is queued in the *unknown-phases* list (shaded box). After the first processing phase in the PE, the total number of phases becomes known, and the packet is queued in one of the lists with a known number of phases.

unit ϕ (step 4 in Figure 3). Ordering unit ϕ checks if one of the N head-of-line packets in its buffer has SN equal to the first (oldest) SN in the list ϕ and whether it is larger than the first (oldest) SN in the list of packets with an unknown number of phases. If the two conditions are met, the packet departs and its SN is removed from the list ϕ (step 5 in Figure 3).

The above algorithms are run-to-completion, i.e. the packet processing is not preempted at any stage, and only a trigger is sent to the data structure in order to update the link list. The required number of processing phases for a packet is determined during the first phase in the PE.

C. Correctness of the RP^3 Algorithms under Frameworks 1 and 2

Theorem 2: The RP^3 algorithm under Frameworks 1 and 2 preserves the order of the packets in each flow.

Proof: Consider packets A and B from the same flow. Assume packet A arrived before packet B to the NP ($t_{arr,A} < t_{arr,B}$, as defined in the proof of Theorem 1). The SN generator will assign SN_A to packet A and SN_B to packet B ($SN_A < SN_B$). Assume that packet B finishes its processing in the PE and is placed in the OU. All possible cases can be divided into the following possibilities for packet A:
Case 1: Packet A is waiting in the PE input buffer. Its processing has not started, thus its number of required processing phases is unknown. Therefore, $\min(SN_u) \leq SN_A < SN_B$, and hence packet B will not depart.
Case 2: Packet A is currently processed in the PE. Its number of

required processing phases ϕ is known. Therefore, $\min(SN_\phi) \leq SN_A < SN_B$ and hence packet B will not depart.
Case 3: Packet A finished its processing but is preempted from transmission because of an earlier packet C. Packet B will not be transmitted because it is also preempted by packet C: ($\min(SN_u) \leq SN_C < SN_B$ or $\min(SN_\phi) \leq SN_C < SN_B$).
Case 4: Packet A finished its processing and has departed. Packet B is not preempted for departure because of packet A. ■

V. RP^3 Algorithm under Framework 3

A. Overview

Under Framework 3, the PEs only know the number of processing phases a packet has already gone through, but not the number of remaining ones.

Intuitively, the problem in Framework 3 is that a packet that has completed its processing by going through ϕ processing phases is blocked from departing the NP by all the packets that have arrived earlier, have completed less than ϕ processing phases, and are still in some PE. This is because all these other packets may, or may not, eventually complete with exactly ϕ processing phases, and therefore potentially belong to the same flow. So the algorithm needs to monitor all these other packets to determine whether the packet is free to go. This intrinsically introduces *three new problems* in the RP^3 algorithm:

First, each packet now needs to go through *several* sequence-number (SN) generators, since it doesn't know its SN generator in advance. For instance, assume a packet needs to complete ϕ processing phases. Since it doesn't know it before completion, it will first ask for the first SN generator; then, for the second SN generator, and so on, gradually discovering how many phases it has.

A second new problem is that *when a packet requests a new SN, it cannot get it automatically anymore*. Consider the following example, which we later extend in Figure 5. Assume that incoming packet A is assigned $SN = 1$ by the first SN generator, and the next incoming packet B is assigned $SN = 2$ by the same first SN generator. Now, assume that packet B completes its first processing phase, but packet A hasn't yet. If B needs to go through a second processing phase and requests an SN from the second SN generator, which one should it get? We do not know yet whether A will also request an SN from the second SN generator, and therefore do not know if the second SN for B should equal 1 or 2. Therefore, A blocks the SN-granting process for B. More generally, the algorithm preserves flow order by making sure to only increase the SN of the oldest packet in the current sequence phase.

A third new problem is that *we want to design our algorithm so that each PE is work-conserving*, i.e. each packet on each PE can *run-to-completion* independently of the sequencing scheme. As a consequence, the packet processing continues even if the new SN is not granted yet. This makes the processing phases and sequencing phases distinct. For instance, a packet may be in the middle of the processing of phase 4, but its sequence number may still belong to phase 2. While this makes the algorithm more efficient, it also makes it significantly more complex to understand.

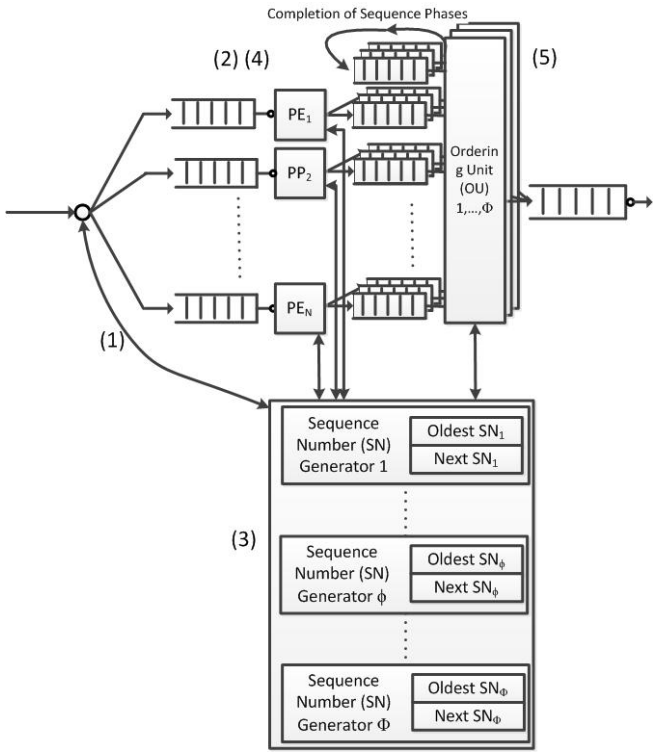


Fig. 4. RP^3 Algorithm under Framework 3. Each packet with ϕ required *processing* phases also passes through ϕ *sequencing* phases. In each sequencing phase, a request token is sent to the SN generator, and a grant token is received. The SN generator sends a grant token only when the requested SN is equal to the oldest SN counter. The generated grant token includes the next SN from the next SN generator. The packet order is preserved for the packets in the same sequencing phase.

We further denote as a *sequence phase* the time between receiving two consecutive SN grants. Note that the final number of sequence phases for a packet is equal to its total number of required processing phases.

Let's now formally describe the algorithm before providing a clarifying example.

B. Algorithm Description

Figure 4 illustrates the RP^3 Algorithm under Framework 3. Assume that the maximum possible number of processing phases per packet is Φ. The architecture includes Φ SN generators, and Φ ordering units (OUs) with $N + 1$ input queues, numbered 0 through N. The ϕ-th SN generator is responsible for preserving the order of all the packets that are currently in the ϕ-th *sequence phase*. The ϕ-th ordering unit holds the packets that have finished processing and are currently within the ϕ-th sequence phase. A packet sequence number is defined by the pair $(\phi : SN_\phi)$, where the SN generator that assigns the sequence number is the ϕ-th SN, and the sequence number that it assigns is SN_ϕ.

Upon arrival, a packet is assigned a sequence number SN_1 by the SN generator 1 (step (1) in Figure 4). It then joins the queue of one of the PEs, as determined by the load-balancer.

After finishing the ϕ-th processing phase of the packet in the PE, a request token with SN_ϕ is sent to SN generator ϕ (step (2) in Figure 4). The SN generator ϕ checks if the sequence number SN_ϕ of the request token is the minimal one, i.e. the oldest sequence number of the packets in sequence phase ϕ. When the condition is met, $SN_{\phi+1}$ is retrieved from SN generator $\phi+1$ and SN_ϕ is released. A grant token with $SN_{\phi+1}$ is sent, and the sequence number of the packet is updated (step (3)).

This completes a single *sequence phase*. When PE i finishes processing the packet, the packet descriptor is sent to the input queue i of the ordering unit ϕ, where ϕ is the number of passed packet sequence phases (step (4)).

At this point, as mentioned, the current number of sequence phases of the packet, i.e. the number of SN re-assignments of the packet, may be smaller than the total number of processing phases it needs to achieve. This is because the packet is waiting for some SN grant token, due to some earlier-arrived packet that has not yet completed processing. As we stated before, the final number of sequence phases has to be equal to the total number of processing phases. Thus, the packet will complete the needed sequence phases in the ordering unit. (Such a case is further illustrated in the example below with packet D.)

For each of the head-of-line packets in its input queues, ordering unit ϕ will send a request token to SN generator ϕ. After receiving the grant token $SN_{\phi+1}$, the packet is pushed to the input queue numbered 0 of ordering unit $\phi+1$. More generally, input queue 0 is reserved for packets that completed their sequence phase in another ordering unit, while input queues 1 to N are reserved for the packets that were pushed from the PEs.

Finally, when the number of passed sequence phases is equal to the total number of processing phases, the packet departs and its sequence number is released (step (5)).

The next example illustrates and clarifies how the RP^3 algorithm works under Framework 3.

Example 1: Figure 5 shows an example of sequence number assignment and the in-order transmission of the packets. Packets A, B, C and D arrive to the NP in this order, and are processed in parallel. Packets A and C require one processing phase ($\phi_A = \phi_C = 1$), and may belong to the same flow. Packet B requires three processing phases ($\phi_B = 3$), and therefore belongs to a different flow. Packet D requires two processing phases ($\phi_D = 2$), and belongs to yet another flow. Of course, in Framework 3, the number of processing phases required by each packet is unknown to the NP until packet processing is completed. The rectangles in the figure present the completion time of each processing phase. For instance, $t_{A,1}$, $t_{B,1}$ and $t_{C,1}$ are the completion times of the first processing phase of packets A, B and C, respectively.

Notice first that the first increment of the sequence number of packet B happens only after incrementing the sequence number of packet A ($t_{A,1}$), and *not* immediately after finishing its first processing phase ($t_{B,1}$). This is because the SN generator cannot assign the next SN to packet B as long as it does not know that packet A will not need it.

In addition, once the processing of packet C completes after one processing phase, it still needs to wait for $t_{A,1}$ and $t_{B,1}$ in order to leave the NP. The reason is that as long as A and B do not complete their first phase, the NP does not know whether A and B need only one processing phase, in which case they may belong to the same flow, or more processing phases, in which case they definitely do not belong to the same flow. Of course, packet C does not need to wait for $t_{D,1}$, because packet D arrived after packet C.

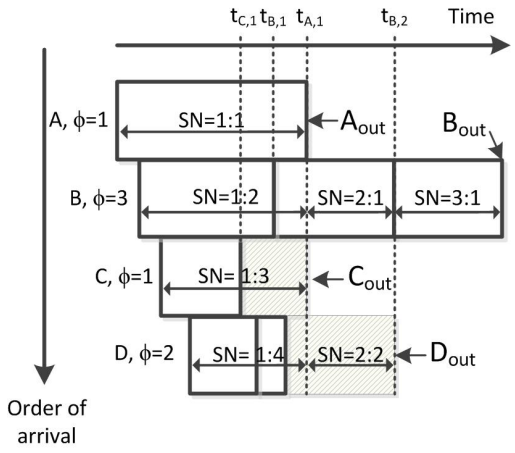


Fig. 5. Sequence Numbering Example. Rectangles present the processing phases, and double arrows illustrate the SN generators and values. A packet sequence number is defined by the pair $(\phi : SN_\phi)$, where the SN generator that assigns the sequence number is the ϕ^{th} SN, and the sequence number that it assigns is SN_ϕ. Packets A, B, C and D are processed in parallel. Packet C can be transmitted only after A and B complete their first sequencing phase at $t_{A,1}$ and $t_{B,1}$. The reordering delay in this case is equal to $t_{A,1} - t_{C,1}$. Packet D completed its processing phases before completing its sequencing phases, and waits for B to complete its second sequencing phase at $t_{B,2}$.

In this case, the reordering delay of packet C is equal to $t_{A,1} - t_{C,1}$. Note that with previously-known reordering algorithms, the reordering delay of C could have been larger and equal to $t_{B,3} - t_{C,1}$, since C would have waited for the processing completion of B. Also note that packet processing is never preempted, even if the next sequence number cannot be received, i.e. the suggested algorithm does not pause packet processing in any case.

Finally, packet D finishes both its processing phases before the increment of its sequence number, and is buffered in the ordering unit. It can be transmitted only after packet B finishes its second sequencing phase at time $t_{B,2}$.

C. Correctness of the RP^3 Algorithm under Framework 3

The next theorem proves the correctness of the RP^3 algorithm under Framework 3.

Theorem 3: The RP^3 algorithm under Framework 3 preserves the order of the packets from the same flow.

Proof: Consider packets A and B from the same flow. Packet A arrived before packet B to NP ($t_{arr,A} < t_{arr,B}$). By assumption, both packets will go through an equal number of total phases Φ. Therefore, their total number of sequence phases is also Φ. If the current sequence phase of packet A is larger or equal to the sequence phase of packet B, packet B will not be transmitted before packet A. Assume at some stage that

both packets completed the same number of sequence phases i, therefore $SN_{i,A} < SN_{i,B}$. Now if packet B tries to advance to the next sequence phase by sending a request token, it will not receive a grant token because $\min SN_i \leq SN_{i,A} < SN_{i,B}$. At the last sequence phase Φ: $SN_{\Phi,A} < SN_{\Phi,B}$, therefore packet A will be transmitted before packet B, and the flow order will be preserved. ■

VI. Performance Analysis Model

To provide more intuition on the efficiency of our RP^3 algorithm, we now analyze its reordering delay and total delay as functions of the traffic arrival pattern and the processing delay distribution. Specifically, we want to compare the RP^3 algorithm against two baseline architectures:

Single SN algorithm: As illustrated in Figure 1, a commonly-implemented and straightforward way to preserve packet order is to use a single global sequence number (SN) generator. Each arriving packet is simply stamped with an incremented sequence number. Then, the reordering unit at the output link transmits only packets with the oldest sequence number. Other packets keep waiting. We will also later show that while this simple architecture is easy to implement, it can cause high reordering delays, because packets of one flow may need to wait for a long time for late packets of a different flow.

Hashed SN algorithm [16]: As discussed in the related work, it is also possible to statically aggregate flows into ordering domains using hashing. We will use this algorithm as our second baseline algorithm, and denote it as *Hashed SN*.

We start by defining the *potential reordering delay* of a packet as the difference between its arrival time, and the latest departure time of a previously-arrived packet that requires order preservation. For example, consider two packets A and B, such that their order needs to be preserved. Assume that B arrives at $t_{\text{arr,B}}$, and A departs at $t_{\text{dep,A}}$. Then the potential reordering delay of packet B is defined as $\max(t_{\text{dep,A}} - t_{\text{arr,B}}, 0)$.

Our goal is to present *the cumulative distribution function of the total delay* T_T, given the distributions of the processing delay (T_{proc}) and of the potential reordering delay (T_{RO}). We make several simplifying assumptions. First, we assume that the processing delay includes both the processing time and the buffering time. We also assume that we can neglect the transmission delay of the packets at the output. We further assume that the processing delay and the potential reordering delay follow independent distributions, and that the processing delays of different packets are independent as well. Finally, we assume a common slotted-time model [29].

The following lemma first describes a general delay model that will be used in the later theorems.

Lemma 1: The distribution of the total packet delay can be modeled as:

$$\Pr(T_T \leq i) = \Pr(T_{\text{proc}} \leq i) \cdot \Pr(T_{\text{RO}} \leq i), \tag{3}$$

where $\Pr(T_{\text{RO}} \leq i)$ depends on the arrival traffic pattern and on the scheduling algorithm.

Proof: A packet B that completes processing at some time t can depart iff there is no earlier packet A preventing it from leaving at t; and if there is such a packet, it can only depart when the last such bottleneck-packet A departs. In other words, the actual departure time of B will be $\max(t_{\text{dep,A}}, t_{\text{dep,B}})$ (where $t_{\text{dep,B}}$ is the depature time of B when no reordering delay occurs, i.e. end of processing of B). As mentioned, we neglect the transmission delay of packet B at the output. Therefore, its total time in the NP will be

$$\begin{aligned} T_T &= \max(t_{\text{dep,A}}, t_{\text{dep,B}}) - t_{\text{arr,B}} \\ &= \max(\max(t_{\text{dep,A}} - t_{\text{arr,B}}, 0), T_{\text{proc}}) \\ &= \max(T_{\text{RO}}, T_{\text{proc}}), \end{aligned}$$

where the second max in the second line is of course superfluous. Therefore, the independence of the random variables T_{RO} and T_{proc} yields the result. ■

The next theorems model the behavior of the algorithms under Bernoulli arrivals with a probability p of packet arrival per time-slot and Poisson arrivals of rate λ. We first analyze the behavior of the Hashed SN algorithm, with m buckets in the hash table. The behavior of the Single SN algorithm is easily derived using $m = 1$.

Theorem 4: Under a Bernoulli-distributed traffic arrivals with a probability p of packet arrival per time-slot, the total packet delay distribution under the Hashed SN reordering algorithm satisfies:

$$\begin{aligned} \Pr(T_T \leq i) = &\Pr(T_{\text{proc}} \leq i) \cdot \\ &\prod_{j=1}^{\infty} \left(1 - \frac{p}{m} \cdot (1 - \Pr(T_{\text{proc}} \leq i + j))\right). \end{aligned} \tag{4}$$

Proof: Following the previous theorem (Lemma 1), an earlier packet arrives at slot $(t - j)$ with probability p, and in that case is only delayed beyond $t + i$ with probability $(1 - \Pr(T_{\text{proc}} \leq i + j))$. The results follows by multiplying all the probabilities that there is no late packet from slot $(t-j)$ over all such possible slots. In addition, due to m uniformly hashed ordering domains, only the packets within the same ordering domain (same hash bucket) will prevent a packet from leaving with probability $\frac{1}{m}$. ■

Theorem 5: Under Poisson-distributed traffic arrivals with total arrival rate of λ packets per time-slot, the total packet delay distribution under the Hashed SN reordering algorithm satisfies:

$$\Pr(T_T \leq i) = \Pr(T_{\text{proc}} \leq i) \cdot \exp\left(-\frac{\lambda}{m} \sum_{j=1}^{\infty} (\Pr(T_{\text{proc}} > i + j))\right). \tag{5}$$

Proof: Similarly to Theorem 4, while now an arrival of several packets per time slot is allowed according to the Poisson distribution:

$$\begin{aligned} \Pr(T_T \leq i) = &\Pr(T_{\text{proc}} \leq i) \cdot \\ &\prod_{j=1}^{\infty} \left(1 - e^{-\lambda/m} \sum_{k=1}^{\infty} \frac{(\lambda/m)^k}{k!} (1 - (\Pr(T_{\text{proc}} \leq i + j))^k)\right), \end{aligned} \tag{6}$$

where $(\lambda/m)^k e^{-\lambda/m}/k!$ is the Poisson-distributed probability for the arrival of k packets in a time slot given a total arrival

rate of λ/m. Using Taylor series function $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ we get:

$$\Pr(T_T \leq i) = \Pr(T_{\text{proc}} \leq i) \cdot \prod_{j=1}^{\infty} \Big(1 - e^{-\lambda/m}(e^{\lambda/m} - e^{\lambda/m \cdot \Pr(T_{\text{proc}} \leq i+j)})\Big). \quad (7)$$

Then,

$$\Pr(T_T \leq i) = \Pr(T_{\text{proc}} \leq i) \cdot \prod_{j=1}^{\infty} \exp\Big(-\frac{\lambda}{m}(1 - \Pr(T_{\text{proc}} \leq i+j))\Big). \quad (8)$$

Finally,

$$\Pr(T_T \leq i) = \Pr(T_{\text{proc}} \leq i) \cdot \exp\Big(-\frac{\lambda}{m}\sum_{j=1}^{\infty}(1 - \Pr(T_{\text{proc}} \leq i+j))\Big) \quad (9)$$

and hence the result. ∎

We now want to model the total delay in the RP^3 algorithm. To do so, we denote $T'_{\text{proc}}(\phi)$ as the sum of the processing delays of the first ϕ processing phases of a packet.

Theorem 6: Under a Bernoulli-distributed traffic arrivals with a probability p of packet arrival per time-slot, the distribution of the total packet delay under the RP^3 reordering algorithm under Framework 1 satisfies:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(\phi = \phi_0) \cdot \Pr(T'_{\text{proc}}(\phi_0) \leq i) \cdot \prod_{j=1}^{\infty} \big(1 - p \cdot \Pr(\phi' = \phi_0) \cdot (\Pr(T'_{\text{proc}}(\phi_0) > i+j))\big). \quad (10)$$

Proof: The distribution of the total packet delay follows:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(T_T \leq i | \phi = \phi_0) \cdot \Pr(\phi = \phi_0),$$

where we denote $Pr(\phi = \phi_0)$ as the probability of the packet to be processed for exactly ϕ_0 processing phases. Thus we are reduced to the same problem, but knowing the number ϕ_0 of processing phases. Then the proof is very similar to that of Theorem 4, but now only the packets with the same number of processing phases will prevent a packet from leaving. The probability of such packets occurring is $Pr(\phi' = \phi_0)$ instead of $\frac{1}{m}$. ∎

Theorem 7: Under a Poisson-distributed traffic arrivals with total arrival rate of λ packets per time-slot, the distribution of the total packet delay with the RP^3 reordering algorithm under Framework 1 can be modeled as:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(\phi = \phi_0) \cdot \Pr(T'_{\text{proc}}(\phi_0) \leq i) \cdot \exp\Big(-\lambda \sum_{j=1}^{\infty} \Pr(T'_{\text{proc}}(\phi_0) > i+j)\Big), \quad (11)$$

Proof: Similarly to Theorem 6, while now an arrival of several packets per time slot is allowed according to the Poisson distribution. Next, knowing the number ϕ_0 of processing phases, the proof is very similar to that of Theorem 5 ∎

Comparing Equations (4) versus (10) and (9) versus (11) reveals the key difference between the previous Hashed-SN approach and our RP^3 approach. While in the former, the packet can be delayed by an arbitrary previously-arrived packet only if it is hashed to the same ordering domain with probability $\frac{1}{m}$, in the RP^3 approach the packet can be delayed only by a previously-arrived packet with similar processing requirements, with a probability $\Pr(\phi' = \phi_0)$.

Finally, we derive the following model for the RP^3 algorithm under Framework 3. Given that the examined packet consists of a total of ϕ_0 processing phases, for each of the earlier packets, the reordering delay occurs if the actual processing of the first ϕ_0 processing phases of one of the earlier packets will last after the completion of the actual processing of the examined packet.

Theorem 8: Under Bernoulli-distributed traffic arrivals with a probability p of packet arrival per time-slot, the total packet delay distribution under RP^3 for Framework 3 satisfies:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(\phi = \phi_0) \cdot \Pr(T'_{\text{proc}}(\phi_0) \leq i) \cdot \prod_{j=1}^{\infty} \Big(1 - p \cdot (1 - \sum_{\phi'=1}^{\infty} (\Pr(\phi_j = \phi') \cdot \Pr(T'_{\text{proc}}(\min(\phi_0, \phi')) \leq i+j)))\Big). \quad (12)$$

Proof: The distribution of the total packet delay follows:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(T_T \leq i | \phi = \phi_0) \cdot \Pr(\phi = \phi_0),$$

where we denote $Pr(\phi = \phi_0)$ as the probability of the packet to be processed for exactly ϕ_0 processing phases. Thus we are reduced to the same problem, but knowing the number ϕ_0 of processing phases. Following Lemma 1, the packet with ϕ_0 processing phases that arrives at slot t can be delayed due to reordering by an earlier packet that arrives at slot $(t-j)$ with probability p, if the processing of the first ϕ_0 phases of the earlier packet last beyond $t+i$. Note that it is possible that the total number ϕ' of processing phases of the earlier packet is less than ϕ_0, therefore we consider $\min(\phi_0, \phi'))$ of the first processing phases of the earlier packet. The result follows by multiplying all the probabilities that there is no delaying packet from previous slot $(t-j)$ over all such possible slots. ∎

Theorem 9: Under Poisson-distributed traffic arrivals with a total arrival rate of λ packets per time-slot, the total packet delay distribution under RP^3 for Framework 3 satisfies:

$$\Pr(T_T \leq i) = \sum_{\phi_0} \Pr(\phi = \phi_0) \cdot \Pr(T'_{\text{proc}}(\phi_0) \leq i) \cdot e^{-\lambda \sum_{j=1}^{\infty} \left(1 - \sum_{\phi'=1}^{\infty} \Pr(\phi_j = \phi') \cdot \Pr(T'_{\text{proc}}(\min(\phi_0, \phi')) \leq i+j)\right)}.$$

Proof: Similarly to Theorem 8, while now an arrival of several packets per time slot is allowed according to the Poisson distribution. Using Taylor series function $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ we get the result. ∎

Incidentally, note that the presented expressions in this section contain infinite products, but all of them converge in practice. In particular, the Poisson probabilities decrease exponentially fast for large k, and the number of phases is typically bounded.

VII. Simulations

A. Simulation Settings

To evaluate our suggested algorithms, we simulate a parallel network processor with $N = 16$ cores. The NP provides a pull-based I/O interface in which incoming packets are stored in a shared input queue serving all the PEs, thus achieving an efficient load-balancing. Related architectures are reported to be implemented in the Cisco QuantumFlow [11] and EZChip NP-4 [9].

We implement our RP^3 algorithm under the three frameworks. We further compare it against our two baseline algorithms, Single SN and Hashed SN. We also compute a reordering delay lower-bound, achieved using an idealized algorithm that would keep a per-flow sequence numbering mechanism. Time is continuous, and reordering delay is measured in time units.

To analyze the performance of our algorithms, we start by using a synthetic traffic arrival pattern. We assume that packet arrivals follow a Poisson distribution. Packets are distributed across 300 flows. The distribution of the flows is assumed to follow a power law (Zipf-distributed with exponent $s = 1$) [38]. For each flow, the number of logical processing phases for all of its packets is chosen uniformly over the $[1, 10]$ interval. Moreover, for a fair comparison, when implementing the *Hashed-SN* algorithm, the flows are hashed into 10 hash buckets. Note that we use a real hash function implementation [39], and not simply a random number generator.

B. No Phase Processing Delay Variability

First we check the performance of our algorithm under ideal settings, without any phase processing delay variability in the PEs. In such an environment all processing phases are equal. Figure 6 illustrates the reordering delay as a function of the load. Our RP^3 algorithms outperform both the baseline algorithms under all three Frameworks. Moreover, the reordering delay is equal to 0, which supports the findings of Theorem 1.

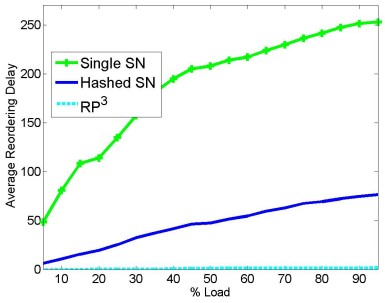


Fig. 6. Impact of load on reordering delay under no processing delay variability. The reordering delay of the RP^3 algorithm is equal to 0 under any load and any framework.

C. Considering Phase Processing Delay Variability

Next we want to verify the impact of the delay variability of the logical processing phases on the performance of our RP^3 algorithms. Intuitively, we would expect our algorithms to perform better when delay variability is low. We add to our simulations a *lower bound* delay graph, which is the reordering delay obtained by a perfect reordering algorithm in which a packet can endure a reordering delay due to other packet of the same flow only. In our simulations, we use two different delay variability models, while keeping the mean processing time for each phase at 100 time units:

Phase Variability: In this model, we assume that the processing time for each phase is uniformly distributed over some interval. Specifically, we define the *phase processing delay variability* as the ratio between the maximal and minimal processing time for a single phase. Each processing phase delay is then uniformly distributed between the minimum and the maximum values, with an average of 100 time units. For instance, a phase processing delay variability of 3 corresponds to a factor of 3 between the minimum and the maximum, i.e. the minimum is 50 and the maximum is 150. A packet with two phases would consecutively draw two such uniformly-distributed random variables.

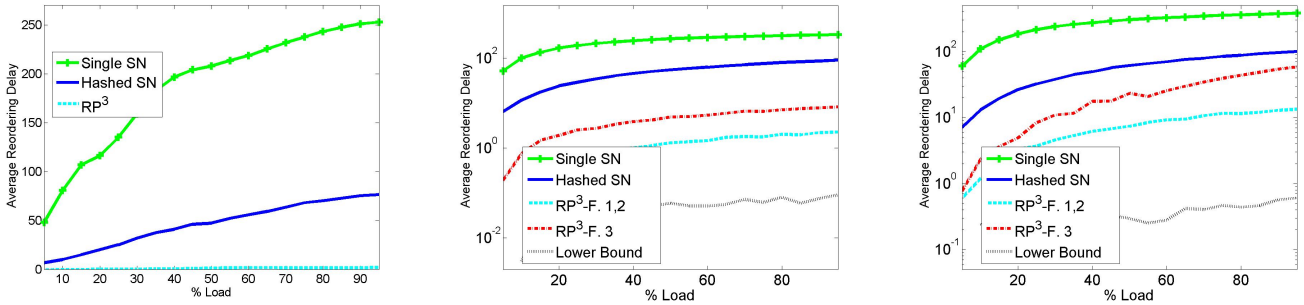
Packet Variability [40]: This model is based on the measurements of [40]. Figure 9 in [40] presents the measured data, and shows a variation that is *constant* for packets with different lengths. In other words, while in our initial model there is more variability for packets with more processing phases, under this new model, the variability is constant. This new model may apply, for instance, if variability essentially results from the initial packet buffering delay, and not from the processing itself. We formalize these measurement results of [40] by defining a packet-processing-delay variability model of variability parameter a, where the total processing delay is uniformly distributed in the range $[mean - a, mean + a]$.

D. Traffic Load

Figures 7 and 8 illustrate the impact of traffic load on reordering delay under the two variability models.

First, in Figure 7, we use a *phase variability model* with phase processing delay variabilities of 1.22, 2 and 10.. As shown, the proposed RP^3 algorithms outperform both baseline algorithms. In particular, the RP^3 algorithms reduce the reordering delay by at least an order of magnitude compared to Hashed SN. Note that the results for RP^3 for Framework 1 and Framework 2 appear near-identical throughout the simulations, and therefore they are presented as a unique algorithm. Finally, the lower bound gets positive values due to the phase processing variability.

Next, Figure 8 illustrates the results under a *packet variability model*. The packet processing delay variability is set equal to $a = 100$. Results are relatively similar. The out-performance of RP^3 appears to hold in the same way under this different model.



(a) Phase variability model with variability of 1.22 (i.e. phase delay in $(90, 110)$

(b) Phase variability model with variability of 2 (i.e. phase delay in $(67, 133)$)

(c) Phase variability model with variability of 10 (i.e. phase delay in $(18, 182)$)

Fig. 7. Impact of load on reordering delay under phase delay variability model.

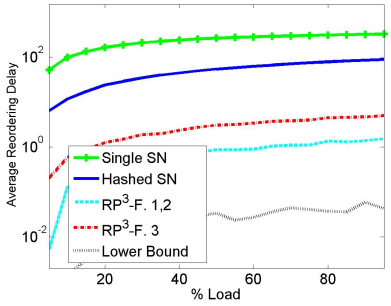


Fig. 8. Packet variability model with variability of 100 time units (model based on the measurements of [40]).

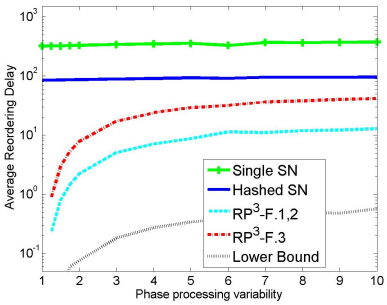


Fig. 9. Impact of delay variability under a load of 90% in the phase variability model.

E. Phase Processing Delay Variability

We then check the performance of the RP^3 algorithms under increasing processing delay variability, using the phase variability model (the results using the packet variability model appear similar). As shown in Figure 9, we vary the phase processing delay variability value from 1x to 10x under a traffic load of 90%. As mentioned before, the correctness of our proposed algorithms is not affected by these variations. However, it is clear that their relative performance is reduced as the variability increases. Still, even with a 10x variability and the unfavourable Framework 3, our RP^3 algorithm interestingly keeps yielding a better result than the two baseline algorithms.

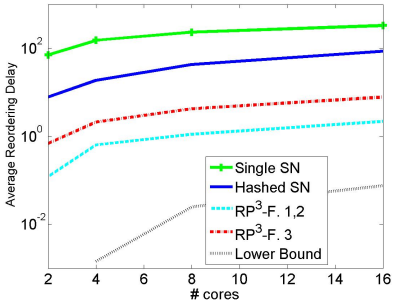


Fig. 10. Impact of number of cores under a load of 90% in the phase variability model with variability of 2.

F. Scaling the Number of Cores

Figure 10 presents the increase of the reordering delay with the number of cores, under the phase variability model (again, the results under the packet variability model appear similar). In the simulation the traffic load is kept at 90% for each number of cores, and the phase processing delay variability is set to 2. Obviously, larger numbers of cores induce more parallel processing and therefore a higher reordering delay. Still, the relative performances of the various algorithms seem insensitive to the degree of parallelism. More generally, note that the scale is logarithmic in all these results, and therefore the improvement is substantial.

G. Real-life Trace Simulations

We next run a set of simulations using a real-life traffic trace from CAIDA [41]. In order to evaluate the processing delay for each packet we assume that each packet is being processed for IP forwarding. We use results from Figure 9 in [40] in order to predict the processing delay of the IP forwarding as a function of packet length. We extrapolate the delay measurements presented in [40] as:

$$\text{Delay}[\text{ms}] = 0.266 \cdot \text{length}[\text{bytes}] + 200 \qquad (13)$$

We use the two variability models. In the *phase variability model*, the PE variation is chosen according to Figures 3(a), (c) and (e) in [40] as approximately equal to 1.22x. Also, following Figure 9 in [40], we use a *packet variability model* with $a = 100$. The mean delay follows Equation (13).

Note that in the trace the lengths of all the packets of the same flow are approximately equal, and therefore by Equation (13) also their processing delays. Thus, when grouping packets by their processing delays, the packets of given flows naturally belonged to the same ordering domains in the simulation, without any need for adding dummy phases. This nicely matched our assumptions without any need for additional tweaks.

Figure 11 presents the simulation results for the average reordering delay as a function of the load, under a phase variability model (Figure 11(a)) and a packet variability model (Figure 11(b)). Figure 11(c) presents the simulation results with no variability in the processing delay. In this case our RP^3 algorithm performs perfectly under all 3 frameworks and all loads. In all simulations our RP^3 algorithm outperforms the compared Single-SN and Hashed-SN algorithms. The similarity between all the plots in spite of the different models suggests a relative robustness of our algorithms.

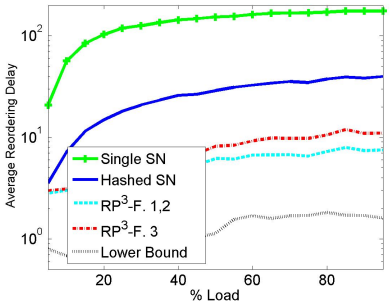
Using these real-life traces, we next study the impact of changing the phase delay variability and of scaling the number of cores in the phase variability model, under a fixed load of 90%. First, Figure 12 shows the effect of varying the phase processing delay variability. As with Poisson traffic, the performance of RP^3 algorithms is reduced as processing delay variability increases. In particular, the out-performance of RP^3 under Framework 3 over Hashed SN disappears for extremely high variability, because classifying flows by the number of processing phases carries significantly less information.

In addition, Figure 13 presents the increase of the reordering delay with the number of cores. Again, the larger number of cores induces a higher reordering delay, yet the relative performances of the various algorithms seems insensitive to the degree of parallelism.

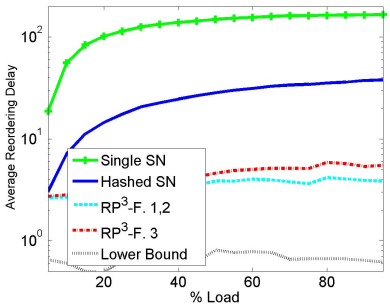
H. Model Evaluation

We also run simulations in order to evaluate our models. The models are checked by simulations with Poisson traffic of arrival rate $\lambda = 1$. The number of phases per packet is assumed to be distributed uniformly and geometrically (with $p = 0.5$) between 1 and 5. In the Hashed SN and the RP^3 algorithms, the flows are distributed among 5 buckets. We validate our models given a phase variability model. We set the phase processing variability to a factor of 3x (phase delay in (50,150)).

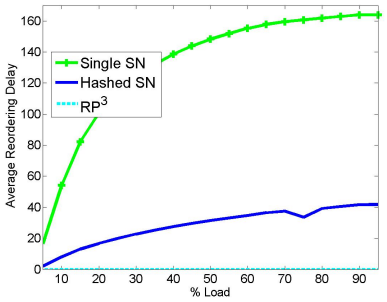
Figures 14, 15 and 16 compare the model and the simulation results for the Single SN, Hashed SN, RP^3 for Framework 1 and RP^3 for Framework 3 algorithms, respectively. The model fits simulations well in all cases. The simulations present some difference with the model due to the fact that in the simulations the arrivals use a continuous time, while the model relies on a discrete-time approximation, as well as due to the model assumptions. Note that the buffering delay appeared negligible in the simulations.



(a) Phase variability model with delay variability of 2 (phase length in (67,133))



(b) Packet variability model with delay variability of 100 time units (based on [40])



(c) Phase variability model with delay variability of 1. The reordering delay of RP^3 is equal to 0.

Fig. 11. Real-life trace: Impact of load on reordering delay.

VIII. Implementation Discussion

A. Complexity of RP^3 Algorithms

We now want to evaluate the additional complexity in our algorithms by comparing them to the baseline *Single SN* and *Hashed SN* algorithms, which were described in Section VI.

Table I illustrates the complexity comparison. We assume a network processor with N cores and Φ buckets in the hashing table.

For the Hashed SN algorithm, we consider an implementation that uses a single SN generator with Φ linked lists, similarly to RP^3 under Framework 1 (as illustrated on Figure 2).

For the RP^3 algorithms under Frameworks 1 and 2, the memory complexity is composed of the ordering units with

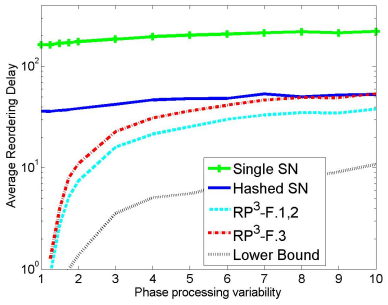


Fig. 12. Real-life trace: Reordering delay vs. phase processing delay variability with load of 90%.

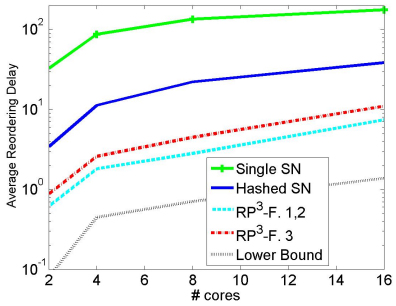
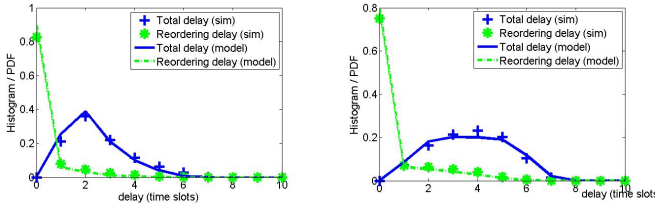


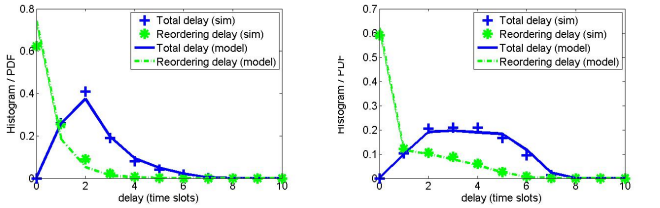
Fig. 13. Real-life trace: Impact of number of cores under a load of 90% in the phase variability model.



(a) Poisson arrivals, geometric processing delay distribution.

(b) Poisson arrivals, uniform processing delay distribution.

Fig. 14. Model vs. Simulation. PDF/histogram of the delay under Hashed SN

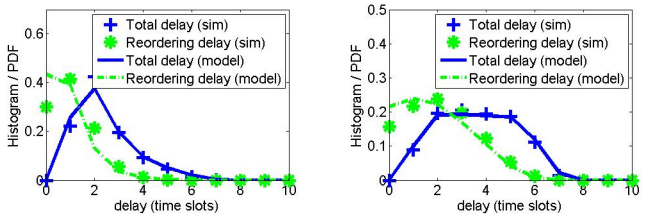


(a) Poisson arrivals, geometric processing delay distribution.

(b) Poisson arrivals, uniform processing delay distribution.

Fig. 15. Model vs. Simulation. PDF/histogram of the delay under RP^3 for Framework 1

$O(\Phi \times N)$ input buffers, and a data structure of $O(\Phi)$ linked lists. Each packet requires $O(1)$ SN generator updates. The additional control bandwidth for a packet is as follows. Assume



(a) Poisson arrivals, geometric processing delay distribution.

(b) Poisson arrivals, uniform processing delay distribution.

Fig. 16. Model vs. Simulation. PDF/histogram of the delay under RP^3 for Framework 3

	Num. of SN generators	Buffers in OU	SN updates per packet	Additional complexity	Internal communication updates per packet (PE-OU) (see text for assumptions)
Single SN	1	N	1		
Hashed SN [16]	Φ	$\Phi \times N$	1		
RP^3 framework 1	Φ	$\Phi \times N$	1		40 bits
RP^3 framework 2	1	$\Phi \times N$	1	Φ linked lists	60 bits
RP^3 framework 3	Φ	$\Phi \times N$	O(Φ)	Φ SN generators	$\Phi * 20$ bits

TABLE I. COMPLEXITY COMPARISON OF THE REORDERING ALGORITHMS

a 16-bit sequence number length and a 4-bit phase identifier. Under Framework 1, the sequence number and phase identifier are transmitted to the data structure twice: For the first time, to assign a SN to the phase list, and for the second time, to release the SN from the list when finally transmitting the packet. Therefore, approximately $20 * 2 = 40$ bits for additional control bandwidth are needed, which is negligible compared to the packet size. Under Framework 2, an additional control bandwidth is required to assign a SN to the unknown-phases list. Therefore, approximately $20 * 3 = 60$ bit of additional control bandwidth is required.

For the RP^3 algorithm under Framework 3, the memory complexity is composed of the ordering units with $O(\Phi \times N)$ input buffers and the data structure of $O(\Phi)$ SN generators. Each packet with ϕ processing phases requires $O(\phi)$ SN generator updates. The additional control bandwidth for each packet is as follows. Assume a 16-bit sequence number length and a 4-bit phase identifier. The sequence number is updated at each sequence phase. Assuming an average of 8 sequence phases for each packet, $8 * 20 = 160$ bits are necessary for additional control bandwidth. Again, this quantity of additional bandwidth is negligible compared to the packet size.

As can be seen in the table, our RP^3 algorithms do not incur significant additional complexity under Frameworks 1 and 2 when compared to the existing Hashed SN algorithm. The complexity overhead increases however under Framework 3, in which there is no *a priori* processing knowledge that can be readily exploited.

It is important to mention that in our proposed algorithms, *only packet descriptors* with the SNs move between the buffers,

and not the full packets with the whole data. Moreover, the phase completion trigger is simple to implement and it do not requires any context switch or process preempting.

B. Combining Hashed SN Algorithm with RP^3 Algorithms

One may notice that the advantages of our approach are reduced for homogeneous traffic, where most of the traffic has similar requirements, and the number of phases for all the flows is equal. This is a limitation of our algorithms. In such cases, our algorithms can be combined with the Hashed SN approach [16]. Thus, the flow partition will be based not only on processing requirements, but also on the flow identifiers.

C. Reducing the Number of SN Generators and Linked Lists

The number of processing phases may vary over a large range of numbers. Thus a large number of linked lists has to be maintained for the algorithm, one for each number of phases. However, in this case, an intuitive improvement is to divide the total potential range of numbers of processing phases into a finite number of smaller ranges. For example, all packets that require a total of ϕ processing phases may be queued in the list of index $\lceil \log_2(\phi) \rceil$.

IX. Conclusion

In this paper, we introduced novel reordering algorithms for parallel multi-core network processors that reduce reordering delays without any meaningful additional cost of implementation. The algorithms are scalable and can be implemented over general-purpose processors. We relied on the fact that all packets of a given flow have similar required processing functions, and therefore that we can divide these into an equal number of logical processing phases. We then introduced three frameworks that define the stages at which the NP learns about the number of processing phases: as packets arrive, or as they start being processed, or as they complete processing. In each framework, we introduced a specific reordering algorithm and provided a theoretical model. Finally, we analyzed these algorithms using NP simulations, and found that reordering delays are negligible, both under synthetic traffic and real-life traces. We also showed how a lower variability in the delays of the logical processing phases leads to significant improvements in the performance of our algorithms.

Acknowledgment

The authors would like to thank Ori Rottenstreich and David Hay for their helpful comments. This work was partly supported by European Research Council Starting Grant No. 210389, by the Hasso Plattner Institute Research School the Intel ICRI-CI Center, and the Israel Ministry of Science and Technology. Support for CAIDA's Internet Traces is provided by the National Science Foundation, the US Department of Homeland Security, and CAIDA Members.

References

[1] P. Paulin, F. Karim, and P. Bromley, "Network processors: a perspective on market requirements, processor architectures and embedded S/W tools," in *DATE, 2001*.

[2] S. Hauger *et al.*, "Packet processing at 100 Gbps and beyond - challenges and perspectives," *Photonic Networks, 2009*.

[3] M. Peyravian and J. Calvignac, "Fundamental architectural considerations for network processors," *Computer Networks*, Apr. 2003.

[4] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," ser. ACM SIGCOMM, 2012.

[5] E. Blanton and M. Allman, "On making TCP more robust to packet reordering," *ACM Computer Communication Review*, vol. 32, 2002.

[6] V. Paxson, "End-to-end internet packet dynamics," *IEEE/ACM Trans. Netw.*, vol. 7, no. 3, pp. 277–292, Jun. 1999.

[7] Cavium, "Octeon II CN68XX multi-core MIPS64 processors [online]," www.caviumnetworks.com/OCTEON-IICN68XX.html, 2010.

[8] AMCC, "NP7310, 10-Gbps network processor with integrated traffic manager [online]," www.datasheetdir.com/NP7310+ Communications-Processor.

[9] EZChip, "EZChip NP-4 network processor, product brief, 2010. [online]," www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf.

[10] Netronome, "NFP-32xx, the first network flow processor for unified computing. [online]," www.netronome.com/pages/network-flow-processors.

[11] Cisco, "The Cisco QuantumFlow processor. product brief, 2010. [online]," www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_ overview_c22-448936.pdf.

[12] Juniper, "Junos trio, white paper, 2009. [online]," www.juniper.net/us/en/ local/pdf/whitepapers/2000331-en.pdf.

[13] N. Weng and T. Wolf, "Pipelining vs. multiprocessors-choosing the right network processor system topology," in *ANCHOR, 2004*.

[14] Z. Cao, Z. Wang, and E. Zegura, "Performance of hashing-based schemes for internet load balancing," in *IEEE Infocom*, 2000.

[15] W. Shi, M. MacGregor, and P. Gburzynski, "Load balancing for parallel forwarding," *IEEE/ACM Trans. Netw.*, Aug. 2005.

[16] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "A hardware packet re-sequencer unit for network processors," in *ARCS*, 2008.

[17] P. He *et al.*, "Balanced locality-aware packet schedule algorithm on multi-core network processor," in *ICFCC, 2010*.

[18] L. Kencl and J.-Y. Le Boudec, "Adaptive load sharing for network processors," in *IEEE Infocom*, 2002.

[19] W. Wu, P. DeMar, and M. Crawford, "Why Does Flow Director Cause Packet Reordering?" *ArXiv e-prints*, Jun. 2011.

[20] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," in *INFOCOM, 2011*.

[21] K. Karras, T. Wild, and A. Herkersdorf, "A folded pipeline network processor architecture for 100 gbit/s networks," in *ANCS, 2010*.

[22] S. Govind, R. Govindarajan, and J. Kuri, "Packet reordering in network processors," in *IPDPS, 2007*.

[23] B. Wu, Y. Xu, B. Liu, H. Lu, and X. Wang, "An efficient scheduling mechanism with flow-based packet reordering in a high-speed network processor," in *HPSR, 2005*.

[24] L. Shi *et al.*, "On the extreme parallelism inside next-generation network processors," in *IEEE Infocom*, 2007.

[25] H. Cheng, Y. Jin, Y. Gao, Y. D. Yu, W. Hu, and N. Ansari, "Per-flow re-sequencing in load-balanced switches by using dynamic mailbox sharing," in *ICC, 2008.*

[26] D. Khotimsky and S. Krishnan, "Evaluation of open-loop sequence control schemes for multi-path switches," in *ICC, 2002.*

[27] F. Chiussi, D. Khotimsky, and S. Krishnan, "Generalized inverse multiplexing of switched atm connections," in *IEEE GLOBECOM, 1998.*

[28] D. Khotimsky, "A packet resequencing protocol for fault-tolerant multipath transmission with non-uniform traffic splitting," in *IEEE GLOBECOM, 1999*.

[29] O. Rottenstreich, P. Li, I. Horev, I. Keslassy, and S. Kalyanaraman, "The switch reordering contagion: Preventing a few late packets from ruining the whole party," *Trans. Computers, to appear*, 2013.

[30] I. Keslassy, "The load-balanced router, ph.d. dissertation, stanford university, june 2004 [online]," http://webee.technion.ac.il/~isaac/thesis/.

[31] T. Wolf, P. Pappu, and M. A. Franklin, "Predictive scheduling of network processors," *Computer Networks*, Apr. 2003.

[32] T. Wolf and M. Franklin, "Locality-aware predictive scheduling of network processors," in *IEEE ISPASS*, 2001, pp. 152 –159.

[33] V. Galtier, K. Mills, Y. Carlinet, S. Bush, and A. Kulkarni, "Predicting and controlling resource usage in a heterogeneous active network," in *Active Middleware Services, 2001.*

[34] F. Sabrina and S. Jha, "Scheduling resources in programmable and active networks based on adaptive estimations," in *IEEE LCN*, 2003.

[35] R. Ramaswamy, N. Weng, and T. Wolf, "Considering processing cost in network simulations," in *SIGCOMM, 2003*.

[36] Ezchip, "Ezchip technologies overview. [online]," www.ezchip.com/technologies.htm.

[37] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, "Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system," in *IEEE Infocom*, 2010.

[38] L. A. Adamic and B. A. Huberman, "Zipf′s law and the internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.

[39] J. Sobel, "JS hash function, [online]," www.partow.net/programming/hashfunctions/index.html#AvailableHashFunctions.

[40] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of network processing workloads," in *Journal of Systems Architecture*, Oct. 2009.

[41] K. Claffy, D. Andersen, and P. Hick, "The CAIDA anonymized 2012 internet traces - (equinix-sanjose/20120119-130000.utc), [online]," www.caida.org/data/passive/passive_2012_dataset.xml.