Flow Migration on Multicore Network Processors: Load Balancing While Minimizing Packet Reordering

Muhammad Faisal Iqbal*, Jim Holt†, Jee Ho Ryoo‡, Gustavo de Veciana§, Lizy K. John¶
*‡§¶University of Texas at Austin
† Freescale Semiconductor Inc. & MIT Computer Science and Artificial Intelligence Laboratory
{*faisaliqbal,‡jr45842}@utexas.edu, {§gustavo, ¶ljohn}@ece.utexas.edu
†jim.holt@freescale.com, jholt@csail.mit.edu

***Abstract*—With ever increasing network traffic rates, multicore architectures for network processors have successfully provided performance improvements through high parallelism. However, naively allocating the network traffic to multiple cores without considering diversified application nature and flow locality results in issues such as packet reordering, load imbalance and inefficient cache usage. Consequently, these issues degrade the performance of latency sensitive network processors by dropping packets or delivering packets out of order. In this paper, we propose a packet scheduling scheme that considers the multiple dimensions of locality to improve the throughput of a network processor while minimizing out of order packets. Our scheduling policy tries to maintain packet order by maintaining the flow locality, minimizes the migration of flows from one core to another by identifying the aggressive flows, and partitions the cores among multiple services to gain instruction cache locality. The scheduler uses a novel low cost two-level caching scheme to identify top aggressive flows. Our light weight hardware implementation shows improvement of 60% in the number of packets dropped and 80% improvement in the out-of-order packet deliveries over previously proposed techniques.**

I. Introduction

Multicore processors are widely used in many application domains. One that greatly benefits from multicore architectures is networking. Network or communications processors used in enterprise, edge and core routers employ highly parallel architectures to meet the demands of diversified applications and ever increasing traffic rates. These routers not only have to handle huge amounts of traffic, but also need to support multiple complex applications like intrusion detection, firewalls, protocol gateways, etc. To meet these requirements, routers need huge processing power. Network processors provide this processing power by exploiting packet level parallelism through a large number of cores. These processors are also software programmable which helps to meet the demands of diverse applications. Many different hardware architectures have been proposed for network processors. These include adapting general purpose multicore processors like Sun Niagara [24] and Tilera [9] or specific architectures for packet processing like Freescale T4240 [2], Broadcom XLP832 [4], EZChip[25], Intel IXP [5] or IBM PowerNP [6]. These specialized communications processors have a large number of cores, in addition to a set of accelerators and co-processors for frequently used network application functions. There have been different designs for network processors, but all of them follow the similar many core approach to exploit packet level parallelism. This huge amount of parallelism has some intrinsic challenges such as load balancing, packet ordering and memory locality.

a) Load Balancing: All cores need to share the load equally to maximize the throughput. An unbalanced allocation of load can overload some cores. As a result, incoming packets will experience large delays and may even result in packet loss due to limited storage in the network processor.

b) Packet Ordering: Performance of upper layer protocols, such as TCP, greatly depends on packet ordering [33]. It is important in applications like VOIP and multimedia transcoding that packets arrive in order because the receiver might not be able to easily reorder the packets. Hence, it is important to preserve the order among packets of a flow. In this work, a flow is a set of packets which have the same source IP, destination IP, source port, destination port and protocol. If packets from the same flow are processed by different cores, they will experience different queuing and processing delays, and consequently, the probability of out of order delivery of packets increases. Careful scheduling of packets is needed in the network processors to minimize out of order departure of packets.

c) Data Locality: If different cores process the packets of the same flow, the data cache will be utilized inefficiently as the same data is copied to multiple caches. There is per flow data (state, statistics), as well as more global data (routing table) used by all flows. If packets of a flow always go to the same core, we can get locality in both local and global data. Locality in global data comes from the fact that different flows may be hot with respect to different parts of the routing table i.e., at the lower levels of the tree. The higher levels are hot to all cores. Furthermore, there are many statistics which are kept per flow, per port etc. Each packet may need to update several of these statistics. If multiple cores work on the packets of the same flow in parallel, this per flow information needs to be kept consistent across these cores by using synchronization primitives like locks or semaphores. This results in blocking access and deteriorates performance. The scheduler needs to account for flow locality in order to get good performance.

d) Instruction Cache Locality: Modern network processors have to support multiple applications but the fast path cores used in these processors are usually small with small I-Cache (8-16KB). These caches can hold only a single program at a time. The performance of a core will deteriorate due to I-cache misses if it has to process packets of different application types.

Researchers have proposed using hashing to distribute packets in parallel network processors [11], [22], [36], [37]. The scheduler hashes one or more header fields of the incoming packet and uses the result to decide the target core for that packet. Packets of the same flow are always mapped to the same core since header fields are constant for all packets of a flow.Hence the flow locality and packet order is maintained. Hash based designs are popular because of simplicity, but they do not perform very well under highly variable traffic conditions and skewed flow sizes. By skewed flow sizes we mean the very common situation where network traffic constitutes several very high data rate flows and very large number of low data rate flows [17], [37]. Furthermore, the hash based schemes need to be adapted for multi-service routers where different packets require different processing and cores are dynamically allocated to services based on traffic variations. Hashing schemes also need to be modified for power saving techniques like [29], [20] which power down the underutilized cores when demand varies. To meet these challenges, this paper makes the following contributions:

1) We propose to partition the cores among multiple services of a router with a separate map table for each service. Each service has exclusive ownership of a subset of cores so that the I-cache locality is maintained. The number of cores allocated to a service changes dynamically with traffic variations. We propose modifications to the hash based schemes for a multi-service router and propose using incremental hashing to manage scheduling when cores are dynamically allocated to services.
2) We present a methodology to achieve load balancing when a core gets overloaded due to skewed flow distribution. This methodology is based on the previous research [37] which proposes to migrate only the aggressive flows to lightly loaded core. Migrating only the top flows achieves load balancing with minimum flow migrations. The scheme proposed in [37] incurs a large overhead since it requires to maintain per flow statistics. Instead of keeping the per flow statistics, we present a novel low overhead scheme based on *annex cache* [21] to identify the top data rate flows.
3) We evaluate the design with real and synthetic traces and show that it can effectively maintain the flow order and cache locality with higher throughput and lower out of order packets than previously proposed schemes.

The rest of the paper is organized as follows: Section II present the background and motivates the problem. The design of scheduler is presented in Section III. Evaluation infrastructure is explained in Section IV and results are discussed in Section V. We present related work in Section VI and conclude in Section VII.

II. Background and Motivation

Architecture for a typical network processor is shown in Figure 1. Incoming packets are received by the Frame Manager (FM). FM places the packet payload in a buffer allocated by the Buffer Manager and places the header, a pointer to the buffer and some meta data as command descriptors in the input queues to the processing cores. These general purpose cores process the packets and can offload some of the work to accelerators e.g., some of the work can be placed in the queue for security accelerator (SEC). SEC performs the required processing and puts it back to the return queue. Eventually, the general purpose core sends the packet back to FM via an enqueue after finishing the processing. Network processing can

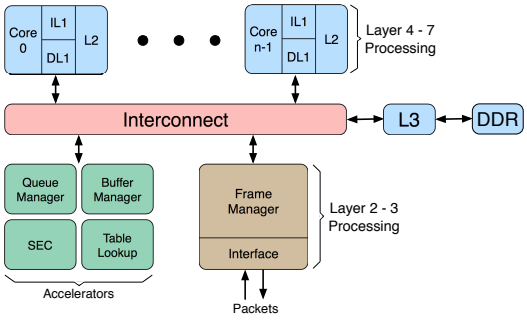


Fig. 1: Architecture of Communications Processors

be classified as either *Control Plane* or *Data Plane*. *Control Plane* is responsible for control and management processing e.g., maintaining and updating the routing tables. *Data Plane* deals with actual processing involved in packet forwarding such as searching, compression, encryption etc. Traditionally, the general purpose cores in the network processor were responsible for processing both control and data plane packets. However, in modern network processors, control and data plane packets take two different paths. When a packet arrives, a packet classifier in the FM decides whether it is a control plane or a data plane packet. Control plane packets take the *slow path* through the general purpose cores. The data plane packets (Layer 2 or possibly Layer 3) take the *fast path* and are not offloaded to general purpose cores. Fast path processing is handled by the FM itself. FM is equipped with a large number (32 - 120) of small cores also called I/O Processors (IOP) which are responsible for fast path processing. These IOPs are typically in-order dual issue cores with non coherent memory, and do not have an operating system. In this work, we are interested in the scheduling of data plane packets on IOPs. We will use the term cores and IOPs interchangeably in rest of the paper.

The design of scheduler for data plane is very challenging. First, the scheduler is in the data path, and therefore, should be as efficient as possible in terms of latency to handle ever increasing traffic rates (100 Gbps and even higher in future). Second, it should satisfy the requirements of load balancing, flow locality, packet ordering and I-cache locality. Hash based designs are popular choices due to their low overhead. These designs only need to compute a hash function

to get the target core for a packet. The packets of the same flow are always mapped to the same core, and hence, the flow locality and packet ordering are maintained. But, there are several challenges associated with the hash based schemes which we try to address in this work. First, it is a well known fact that network traffic constitutes only few heavy-hitter flows and a very large number of low data rate flows [17], [37]. Figure 2 demonstrates this behavior in real network traffic. Under this situation, hashing alone cannot achieve load balance effectively as shown by Shi et al. [37] and can result in overloading some cores. In this scenario, the load of each core should be monitored and adjusted dynamically to migrate some load to underutilized cores. Care must be taken since we want to minimize the number of flow transfers. Previous research has shown that the load can be balanced effectively with minimum flow disruption by migrating only the top aggressive flows [37]. We follow the same approach in dealing with load imbalance. However, the results in [37] are based on off-line analysis and require keeping per flow statistics to identify aggressive flows. Maintaining per flow statistics has significant overheads, and thus, is not feasible in realistic designs. Although many per flow statistics are available in software, it is very time consuming for hardware scheduler to access these software statistics. The scheduler needs to function with minimum software intervention for good performance. We present a novel and low overhead hardware technique to identify aggressive flows by using *Annex Cache*. The annex cache based scheme readily integrates with the scheduler and can be directly accessed. We explain the design of aggressive flow detector in detail in Section III.

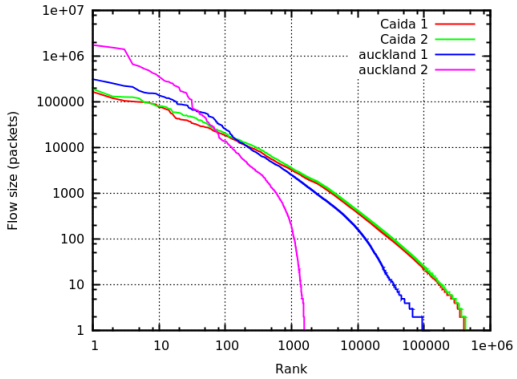


Fig. 2: Distribution of flow sizes in real network traces. Rank 1 is the flow with the largest flow size.

Another challenge associated with hash based schemes is present in multi-service routers. These routers have to support services like IP forwarding, intrusion detection, IPSEC encryption, IPSEC decryption, etc. The mix of packets destined for each service varies with time. If packets of different services are sent to the same core, I-cache locality cannot be maintained which results in huge performance overhead [38], [23], [39]. Therefore, it is necessary that cores are partitioned effectively among services to get high cache locality. Traditionally, cores are allocated to services statically at design time based on their worst case arrival rates. This results in unnecessary hardware over-provisioning with high system cost. All services do not experience their worst case traffic at the same time, so most of the processing resources under-utilized. A system that can multiplex cores among different services fundamentally lowers the number of cores needed and reduces system cost. In this work, we extend the hash based schemes for multi-service routers and propose incremental hashing to manage the scheduling when cores are dynamically allocated to services.

III. Locality Aware Packet Scheduler

In this section we present the design of our proposed Locality Aware Packet Scheduler (LAPS). The design goals of LAPS are: a) To achieve high throughput by maintaining I-Cache and flow locality. b) To minimize out of order departure of packets. c) To have a low overhead in order to sustain high packet rates.

LAPS uses a hash based design which is a natural way of maintaining flow locality. When a packet arrives, its flow identifier is extracted from the header. Flow identifier is a five tuple consisting of source and destination IP addresses, source and destination ports and protocol ID. This five tuple is hashed using CRC16 to get an index into a map table. CRC16 is shown to provide good performance for hashing IP headers [8]. The map table stores target core ID where the packet is eventually forwarded. The hash table based designs are simple but they suffer from several challenges which LAPS tries to overcome. First, in the presence of skewed flow size distribution as shown in Figure 2, LAPS presents an efficient scheme for identifying and migrating aggressive flows. Second, LAPS modifies the hash based design to make it suitable for work in a multi-service router.

A. Load Balancing by Migrating Aggressive Flows

When a core becomes overloaded i.e., its queue size reaches a threshold, the scheduler needs to migrate some of the incoming traffic from that core to a less loaded core. This migration of flows has two drawbacks: One, it makes some cached data in the source core useless and triggers some cold misses in the cache of newly allocated core. Two, flow migration makes it harder to maintain the order among packets of the flow. The new incoming packets will potentially experience less queuing delay as compared to older packets which are waiting in the overloaded core's queue. To avoid these two drawbacks, it is desirable to minimize the number of flow migrations. If we can identify and migrate only the most aggressive flows, load balance can be achieved with minimum disruption i.e., only a few flows need to be migrated to achieve load balance. This is based on the observation made in [37], which indicates that load imbalance is usually caused by a few aggressive flows. However, the scheme proposed in [37] keeps stats for each active flow in order to identify the aggressive flows. This requires a lot of overhead and is infeasible in the practical designs. In contrast, LAPS proposes a novel cache based hardware called Aggressive Flow Detector (AFD) to identify the top flows. The hardware consists of a very small fully associative cache called Aggressive Flow Cache (AFC). AFC is augmented with a cache assist called annex cache. Detailed

architecture of annex cache and AFC is presented in Section V-B. Flows that hit in the AFC are considered aggressive flows. When load imbalance is observed, the scheduler checks if flow ID of the incoming packet hits in the AFC. If it hits, the flow ID is copied into the migration table and is allocated to the least loaded core. Listing 1 shows how the aggressive flows are migrated to the least loaded cores when load imbalance is observed. The scheduler gives priority to the output of migration table over the default hash table. If all the cores get overloaded, this means that the current allocation of cores to this service is not enough to handle the input traffic. An additional core is requested at this instance.

```
1  for (every incoming packet){
2     if (load_imbalance){
3        minq = findMinQ();
4        if (minq < high_thresh){
5           hit = AFC.access(flowID);
6           if (hit){
7              migration_table.add(flowID,minq);
8              AFC.invalidate(flowID);
9           }
10       }
11       else{
12          request_core();
13       }
14    }
15 }
```

Listing 1: `Load Balancing by Flow Migration`

B. I-Cache Locality

A simple hash based design as proposed in [11], [37] can result in inefficient I-Cache usage. It can schedule packets of different applications to the same core. The core will experience a large number of I-Cache misses which will adversely affect the performance. In order to avoid these I-Cache misses LAPS partition the map table among different services i.e., each partition will have its own map table. All the cores in a single map table will always get packets which require the same processing so I-Cache locality will be preserved. LAPS presents strategies to dynamically allocate different number of cores to services based on the traffic and presents methodology for maintaining the map tables of each service under this dynamic allocation of cores with minimum disruption. Figure 3 shows the overall architecture of LAPS.

C. Allocation of Cores to Services

At initialization, cores are equally divided among services. As traffic varies over time, requirement of each service changes and the core allocation needs to be modified. This situation arises when `request_core()` is called in Listing 1. LAPS needs to find an additional core to fulfill the demands of a requesting service and update the map table accordingly. LAPS keeps a list of cores which are marked as surplus cores by other services (Section III-D). When a service requests an additional core, LAPS looks through the list of surplus cores and finds the core which has been marked extra for the longest period of time and allocates this core to fulfill the demands of requesting service. This policy makes sure that the deallocated core has the least utility for the victim service. The core ID is added to the list of allocated cores for the requesting service. In order to minimize the number of flows migrated on core allocation we make use of *Incremental Hashing*. Initially each service has *m* entries in the map table and the hash function used is $h_1(k) = CRC16(k)\%m$. When an additional core is allocated to that service the hash function changes to $h_2(k) = CRC16(k)\%2m$ such that the flows which were initially mapped to index 0 are now divided between index 0 and m. Let b be the current number of buckets in use then our hash function is defined as

$$h(k) = \begin{cases} h_2(k) & : (h_1(k)) < (b-m) \\ h_1(k) & : (h_1(k)) \geq (b-m) \end{cases}$$

The hash function remains the same when more cores are allocated i.e., b increases until b reaches 2m. In that case the second hash function is modified to $h_2(k) = CRC16(k)\%4m$. Use of this incremental hashing in conjunction with load balancing scheme of Section III-A allows us to add additional cores to a service with minimal disruption to the existing flows.

D. Release of Cores by Services

When input queue to a core becomes empty, a timer starts. When the timer reaches $idle_{th}$, the core is marked surplus by adding it to a list of extra cores. The core still remains allocated to the same service. In case, the same service needs more resources in near future, this core can be unmarked and removed from the list of surplus cores without incurring the overhead of context switch. If the core is actually allocated to another service, it is removed from the bucket list of the victim service. Other core IDs will be shifted to take the place of this ID. The bucket size b is decremented by 1 and the hash function is also changed accordingly. This may result in some flow migrations but the performance overhead is tolerable because this service is only lightly loaded anyway.

E. Overall Scheme

Figure 3 shows the overall architecture for LAPS. The bucket list in the mapping table for each service S_i is dynamic and the dynamic size b_i changes with traffic variations. The hash function for each service is decided based on the size of its bucket list. Following steps are taken when a packet arrives:

1) If the flow ID hits in the migration table, the packet is forwarded to the core ID indicated by the migration table.
2) If the flow ID does not hit the migration table, the map table is searched using the hash function and the packet is forwarded to the core indicated by the mapping table.
3) Under load imbalance, the aggressive flows (flows that hit in AFC) are migrated to the least loaded core allocated to that service
4) When number of cores allocated to a service become insufficient, the bucket lists are updated. An idle core is removed from the bucket list of donor service and is added to the bucket list of overloaded service.

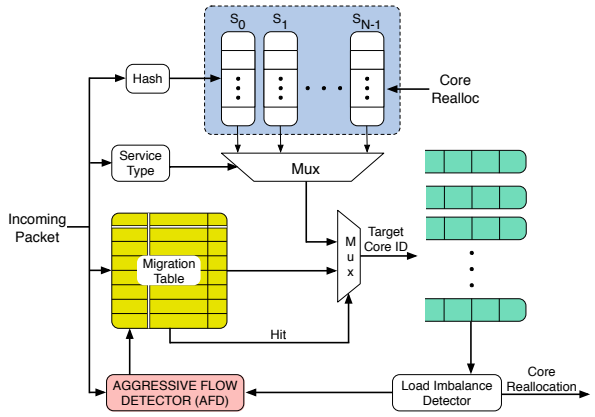


Fig. 3: Locality Aware Packet Scheduler

F. Aggressive Flow Detector (AFD)

The design of Aggressive Flow Detector is based on *annex cache*. Annex Cache was proposed by John [21] to exploit locality in the memory references in general purpose processor workloads. We show that such a structure can be very useful in order to identify aggressive flows. The AFD has two main components as shown in Figure 4. One component is a small fully associative cache called Aggressive Flow Cache (AFC). AFC holds the IDs of top aggressive flows. All entries into AFC come via annex cache. Items referenced only rarely will be filtered out by annex cache and will never enter AFC. The basic premise is that a flow deserves to enter AFC only if it proves its right to be in AFC by showing locality in the annex cache. Annex cache also serves as a victim cache and provides some inertia before a flow is excluded from the AFD. Both AFC and annex cache use Least Frequently Used (LFU) replacement policy. The design of AFD is slightly different from the one presented in [21] because our annex cache is bigger than AFC. We do not want a larger AFC because we want only the aggressive flows to cache in AFC. Annex cache is a bigger structure which serves as a qualifying station for large number of flows to show their eligibility to be cached into the AFC. When a packet arrives, its flow ID is checked in both AFC and annex cache. If it is a hit in AFC, the hit counter is incremented. On a hit in the annex cache, flow counter is incremented and the value is compared with a predefined threshold. If the hit count exceeds the threshold, the flow is promoted to AFC. The victim flow from AFC is then placed in the annex cache. Finally on a miss in annex cache, a flow replaces the LFU flow of the annex cache.

G. Timing Analysis of LAPS

In order to sustain a traffic of 100 Gbps, the scheduler has to be able to schedule 100 Million packets per second (considering mixed sized packets). Note that the critical path of LAPS is Hash Delay → Map Table Access → Mux Delay. AFD and map table update are not part of the critical path since they work in the background. The critical path is dominated by hash Delay. Even the FPGA implementations of CRC16 show that it can operate in excess of 200 MHz [1]. The delay of map table is a fraction of a nano second according to our Cacti

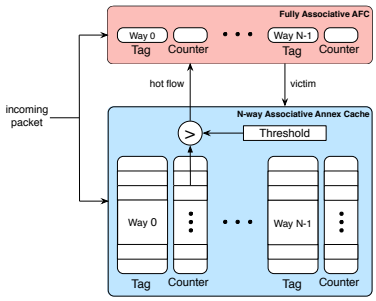


Fig. 4: Structure of Aggressive Flow Detector

[31] simulations. This means LAPS is capable of sustaining at least 200 Million packets per second and more efficient ASIC implementations of hash functions make this design scalable for future traffic of beyond 100 Gbps.

IV. Evaluation Infrastructure

In this section, we describe the evaluation infrastructure and provide a brief introduction of the set of traces used in this study.

A. Traffic Traces

In this work we used real network traces to evaluate the performance of packet scheduler. Following is a small description of set of traces used in this study.

1) CAIDA Traces: This dataset contains anonymized traffic traces from CAIDA's equinix-sanjose monitor [10]. This monitor is connect to OC-192 link. These set of traces are captured in year 2011 and are of duration of 1 minute each.

Trace	Name
Caida 1	20110120-125905.UTC.anon.pcap.gz
Caida 2	20110120-130000.UTC.anon.pcap.gz
Caida 3	20110120-130100.UTC.anon.pcap.gz
Caida 4	20110120-130200.UTC.anon.pcap.gz

TABLE I: List of CAIDA traces used in the study

2) University of Auckland Traces: This set of traces, also known as AUCK-II, is captured at University of Auckland and captures the traffic between the university and its ISP [3]. All connections from the university to external world pass through this measurement point. These traces are of one hour long duration each.

Trace	Name
Auckland 1	20000614-181539-0.gz
Auckland 2	20000614-181539-1.gz
Auckland 3	20000619-183717-1.gz
Auckland 4	20000621-105006-0.gz
Auckland 5	20000621-105006-1.gz
Auckland 6	20000630-175712-0.gz
Auckland 7	20000630-175712-1.gz
Auckland 7	20000703-152100-0.gz

TABLE II: List of Auckland-II traces used in the study

B. Workload Model

In order to model the different services running on a multi-service router, we consider a workload similar to the one presented in Figure 5. This model is based on methodology presented in [19]. In modern network processors, all tasks of the same path are scheduled on the same core to reduce the communication overhead. Hence, in this study we consider all the tasks on the same path as a single service. Thus our simulations have four active services in the processors. A packet is tied to a single core for the life time of its processing. The incoming packets can be serviced by one of the four

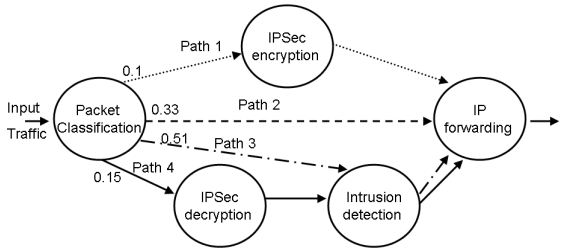


Fig. 5: Example Task graph for an edge router

services represented by different paths of Figure 5. *Path 1* describes the path of outgoing packets which are tunneled via VPN. *Path 2* represents the default handling of packets. *Path 3* is the path of incoming packets on edge router that are scanned for malware and *Path 4* is for incoming VPN packets which are decrypted and scanned for malware.

C. Simulation Infrastructure

For evaluating different scheduling strategies, we developed a simulation model in SpecC [14]. SpecC is similar to systemC [15] in its design and philosophy. Different components of the simulator are shown in Figure 6.

1) Packet Generator: *Packet Generator* generates traffic with programmable traffic rates. To generate packets, it reads the real packet traces. We govern the traffic for each path based on Holt-Winterz forecasting as suggested in [7]. The traffic rate is governed by equation 1.

$$x_i(t) = a + b.t + C.S(t\%m) + n(\sigma) \tag{1}$$

where $x_i(t)$ is the traffic rate for service i, a is the baseline traffic component, b is the trend component, C is the magnitude of seasonal component S, m is the period of seasonal component, n is random noise with a standard deviation of σ. Total incoming traffic is the sum of traffic of each individual service i.e.,

$$X(t) = \sum x_i(t) \tag{2}$$

The header for each generated packet is taken from real network traces. We use a separate packet trace for each path of the flow graph. The use of real network traces ensures that realistic flow scenarios are created.

2) Scheduler: The *Scheduler* module implements the different scheduling strategies. Once a decision has been made the input packets are enqueued into the input queue of the target core. The queue size is set to 32 packet descriptors for each queue based on previous research [32]. A packet is lost when it is assigned to a queue which is already full.

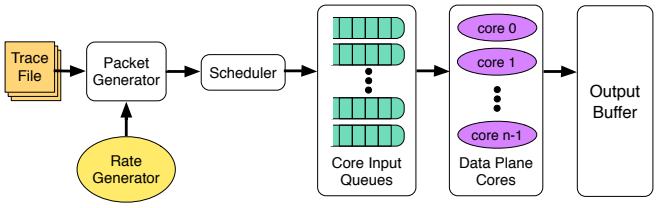


Fig. 6: Simulation infrastructure

3) Processing Latencies: Each packet of a service i, experiences a Processing Delay (PD_i) in the core based on the following equation

$$PD_i = T_{proc,i} + FM_{penalty} + CC_{penalty} \tag{3}$$

Where $T_{proc,i}$ is the processing time, $FM_{penalty}$ is the penalty due to flow migration and $CC_{penalty}$ is the cold cache penalty which occurs when subsequent packet needs different processing than the previous packet. $T_{proc,i}$ is derived from real delays seen by the packets when the packet processing is implemented in software on a full system GEMS [30] simulator. The configuration of in-order cores is shown in Table III.

Frequency	Pipeline	Branch Predictor	I-Cache	D-Cache
1GHz	7 stage 2-issue	gshare/BTB 128 entry each	16KB 2 way	32KB 4 way

TABLE III: Data plane core configuration

We executed these packet processing applications and derived a packet processing delay model for each service. T_{Proc} is measured to be 0.5μs for path 2 i.e., IP forwarding. For path 3, it is measured to be 3.53μs. For Path 1, it also depends on the packet size and is given as

$$T_{proc,path1} = 3.7\mu s + \frac{PacketSize}{64byte} \times 0.23\mu s \tag{4}$$

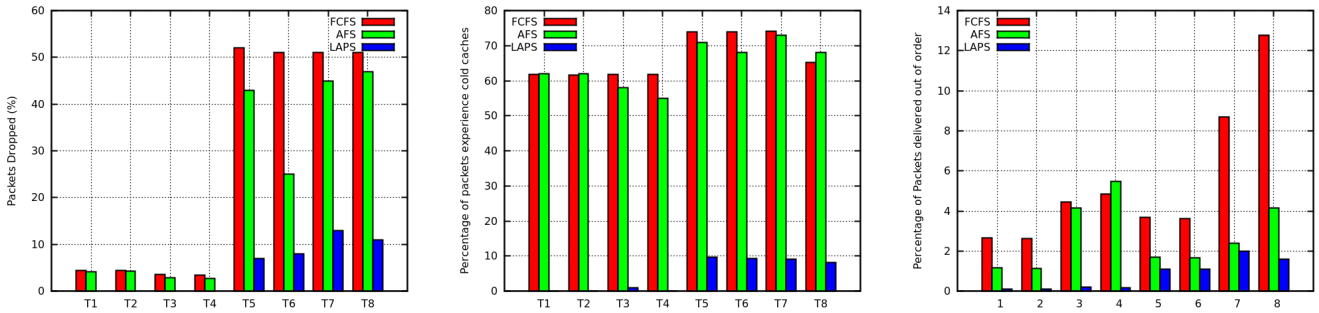
Similarly the processing time for path 4 is given as

$$T_{proc,path3} = 5.8\mu s + \frac{PacketSize}{64byte} \times 0.21\mu s \tag{5}$$

$FM_{penalty}$ is set to four cache misses (o.8μs) conservatively (two for routing data and two for per flow data). In reality, a flow migration can cause a lot more misses depending on how much per flow data is maintained. Because of small I-cache, these cores can hold instructions of only the last executed program (e.g., AES encryption used in IPSec requires 16KB). So whenever a packet of different service arrives at a core, it will experience cold cache penalty. We set the cold cache penalty to 10μs which is the cold cache penalty for the smallest service i.e., IP Forwarding. In practice this penalty will be higher because many services are larger and a context switch can result in some D-Cache misses too. For simplicity, we ignore these data misses due to context switch in this work.

V. Results and Discussion

In this section, we demonstrate the effectiveness of LAPS through experimental results. First, we show that LAPS is able



(a) Packets Dropped (b) Percentage of Cold Caches (c) Out of order packets

Fig. 7: Comparison of LAP with FCFS and AFS with different traffic scenarios listed in Table VI

to handle dynamic traffic variations effectively and exploits I-cache and flow locality to improve the throughput while minimizing packet reordering. Second, we show that our proposed AFD mechanism assists LAPS to identify the top aggressive flows accurately and helps to achieve load balance with minimum flow migrations.

A. Performance Improvement with LAPS

In this section, we compare the performance of LAPS with a First Come First Served (FCFS) scheduler and the scheduler presented in [11]. This scheme migrates arbitrary flows when load imbalance is detected. We call this as Arbitrary Flow Shift (AFS). For this set of experiments, traffic rate is governed by equation 1. We experimented with different sets of parameters for equation 1 and LAPS outperforms other schemes in all scenarios. We present results with two sets of parameters listed in Table IV. Set 1 represents the under-load scenario i.e., the aggregate traffic rate is less than the ideal capacity of 16 cores. Set 2 represents an overload scenario i.e., the data rate is more than the capacity of the 16 core system.

	Service	a	b	C	m	σ
Set 1	S1	1	0.03	0.3	40	0.1
	S2	1.8	-.025	0.1	25	0.05
	S3	0.5	0.01	0.07	60	0.25
	S4	0.3	0.005	0.09	600	0.3
Set 2	S1	1.5	0.002	0.3	100	0.3
	S2	1.3	-.02	0.15	25	0.05
	S3	1	0.004	0.25	30	0.25
	S4	0.7	0.01	0.18	200	0.3

TABLE IV: Parameters governing traffic rate. Rate is in Mpps and period is in seconds

For each service, we use real network traces listed in Table V to generate the packet. The combination of sets of equation in Table IV and traces in Table V creates different traffic scenarios listed in Table VI. Figure 7(a) shows packets dropped with three schemes under the traffic scenarios shown in Table VI for a traffic of 60 seconds. LAPS outperforms FCFS and AFS in both the under-load and overload conditions. FCFS and AFS distribute packets of different services arbitrarily to cores and suffer from poor I-cache locality (Figure 7(b)). These schemes drop packets even in under-load conditions because

Group	S1	S2	S3	S4
G1	Caida1	Caida2	Caida3	Caida4
G2	Caida5	Caida6	Caida2	Caida3
G3	auck1	auck2	auck3	auck4
G4	auck5	auck6	auck7	auck8

TABLE V: Traces used in experiment for packets of individual services

Scenario	Parameter Set	Trace Group
T1	Set 1	G1
T2	Set 1	G2
T3	Set 1	G3
T4	Set 1	G4
T5	Set 2	G1
T6	Set 2	G2
T7	Set 2	G3
T8	Set 2	G3

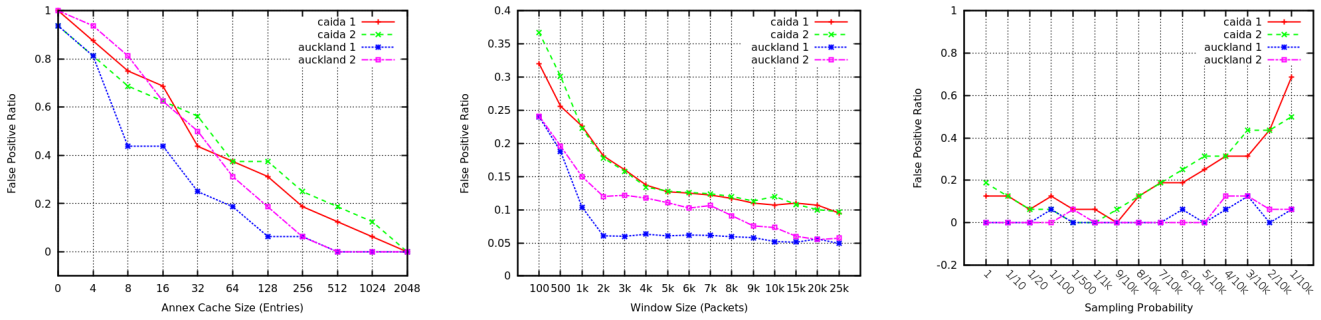
TABLE VI: Traffic scenarios used in Figure 7

almost 60% of packets suffer from cold cache penalties. On the other hand, LAPS partitions the cores among services effectively and enjoys good I-Cache performance. Under overload scenarios (T5 through T8), LAPS also suffers from some cold caches because cores are dynamically switched between services based on traffic variations.

LAPS maintains data and instruction cache locality and is able to sustain higher traffic input rates. Figure 7(c) shows the effectiveness of LAPS in preserving packet order under traffic scenarios of Table VI. FCFS does not care for packet ordering and hence results in the highest out of order packets. AFS improves packet order a little but still there are considerable number of out of order packets due frequent flow migrations. LAPS minimizes the flow migrations by only migrating the top flows and hence results in a very few packets being delivered out of order. Next, we show how our proposed Aggressive Flow Detector (AFD) identifies the top flows and helps to achieve load balance with minimum flow migrations.

B. Performance of Aggressive Flow Detector (AFD)

Recall from Section III that our proposed AFD has two components: An aggressive flow cache (AFC), and an annex cache. An annex cache can be viewed as a preliminary filter



(a) False Positive Ratio in a 16 entry AFC when Annex Cache size is varied

(b) Effect of window size on accuracy of AFD

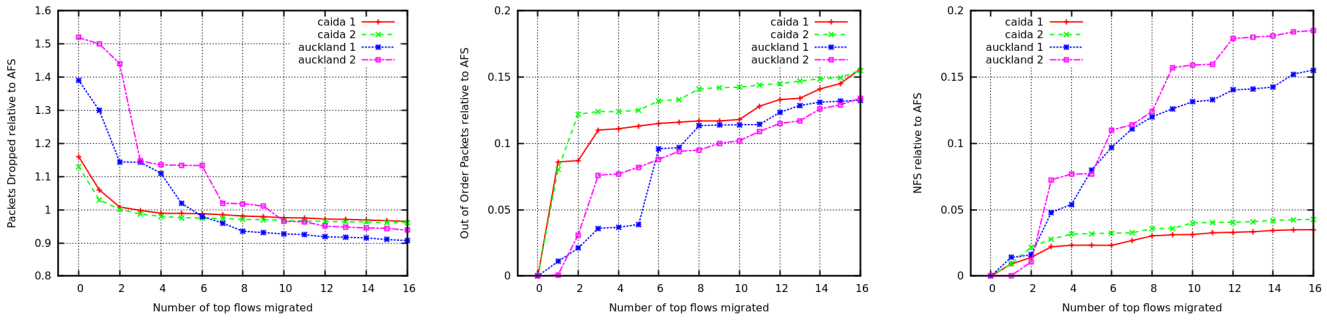
(c) Effect of packet sampling on performance of AFD

Fig. 8: Effectiveness of AFD in identifying aggressive flows

where non-aggressive flows are filtered out from entering the small AFC. Therefore, any entry in AFC is considered an aggressive flow. We evaluate the effectiveness of AFD by varying annex cache size while setting the size of AFC constant at 16 entries. Since our AFC size is fixed, we can only detect up to maximum of 16 top aggressive flows. A perfectly accurate AFC will hold the IDs of top 16 aggressive flows. A flow found in AFC, which is not among the top 16 flows identified by off-line analysis is considered a false positive. Figure 8(a) shows the false positive ratio (false positives/total entries) in AFC when annex cache size is varied. As the size increases, the annex cache can hold more flows to choose a possible candidate for promotion to the AFC. In other words, the pool of aggressive flow candidates increases and the chances of aggressive flows residing in the cache for the AFC promotion becomes higher. For Auckland traces, AFC can identify all top 16 flows with 100% accuracy with a 512 entry annex cache. Caida traces have much more active flows and thus require a larger annex cache. In Caida 1 and 2 respectively, only 14 and 13 most aggressive flows are correctly identified with a 512 entry annex cache. When we double the size to 1024 entries, accuracy improved an average of 6.25%. Although there are 2 or 3 false positives in Caida 1 and 2 cases, they are not random flows that are promoted to the AFC. In fact, when we consider 20 most aggressive flows as our area of interest, these false positives fall into the aggressive flow category. Yet, for consistency of our work, we treat those flows as false positives. We only looked at the accuracy of our mechanism at the end of our simulations until now. Since LAPS needs to peek into the AFC whenever load balancing is required, we performed another experiment where the accuracy is checked at every fixed interval. In Figure 8(b), we performed the same accuracy evaluation with varying interval steps. In this experiment, we fixed the size of annex cache to 512 entries. AFD shows above 90% accuracy from a small step size such as every 1000 packets to large step sizes. This implies that our AFC will contain the most aggressive active flows regardless of when it is accessed. In dynamic scheduling schemes like ours, it is key to maintain a high level of accuracy across the entire execution. Figure 8(c) shows the false positive ratio when packets are sampled with a probability p and not all of them access the AFD. It is interesting to note that FPR improves initially with sampling. This is because sampling acts as a filter i.e., the probability of large flows being sampled is higher than the smaller flows. However, the performance deteriorates for Caida traces at larger sampling intervals. Sampling up to 1/1k probability gives better or equal performance than sampling all packets for all traces. Caida traces generally have a large number of high data rate flows and hence their performance deteriorates if sampling is increased too much. Sampling not only improves the accuracy but also reduces power consumption because now each packet does not have to access the AFD.

C. *Benefit of Limiting Migration to Only Top Flows*

In this section, we demonstrate the benefit of migrating only the most aggressive flows to achieve load balance. We present results relative to the AFS scheme. To demonstrate the effectiveness of LAPS, we simulated a situation where only one service (IP forwarding) is active in the processor. Real network traces are used as input traffic to simulate the real flow scenarios. The input packet rate is set to slightly more than 100% of what this configuration can achieve under ideal conditions. We simulate 60 second traffic and the results are presented in Figure 9. Figure 9(a) shows packets dropped relative to AFS. A lot more packets are lost if we do not migrate any flows, but for almost all traces we can achieve similar or better throughput than AFS if only top 10 flows are identified and migrated. The real benefit of LAPS, however, is to maintain the order of packets. Figure 9(b) shows that the percentage of out of order packets is reduced by 85% if we identify and migrate only the top 16 flows. This benefit comes from minimizing the number of flow migrations as compared to AFS. In AFS, many non-aggressive flows are migrated which incur flow migration penalty without providing any benefit in load balancing. In contrast, if we migrate only the most aggressive flows, we can achieve load-balancing by migrating only a small number of flows and thus can reduce out of order delivery of packets. Figure 9(c) shows that the number of flow migrations are reduced by 80% if we migrate only the most aggressive flows.



(a) Packets Dropped (b) Out of order packets (c) Number of Flow Migrations

Fig. 9: Effect of migrating top flows compared with migrating arbitrary flow (AFS) for load balancing. All plot are relative to AFS

VI. Related Work

Detecting and monitoring aggressive flows is an important part of traffic management and policing. Consequently, there has been plethora of work on how to calculate flow statistics. Initial naive proposals to keep counters for each flow [34], [13] are not scalable when there exist millions of flows, which is common in today's network environment. There have been extensive researches on reducing the overheads of keeping per flow counters [27], [18], [12], [41], [40] to find the accurate estimate of the rates of aggressive flows. In contrast, LAPS merely needs to identify the top aggressive flows without accurately estimating the rates of all flows. The closest to our work is done by Yi et al. [28] where a single cache is used to identify "elephant" flows. Our experiments show that such a scheme can result in large number of false positives due to many "mice" flows active at any time. Our proposed AFD eliminates these false positives effectively by using two-level cachine and integrates directly with the scheduler.

Dittman [11] proposes a hashing based scheme to reduce scheduling decision overheads. This scheme migrates arbitrary flows on load imbalance and can result in large number of flow migrations and out of order packets. This scheme is called Arbitrary Flow Shift (AFS) in this paper. Dittman's scheme is modified by Shi et al. [37] who propose to only migrate the flows which have high data rates. Since Internet traffic has a majority of flows with low activity and very small number of flows with high activity, this scheme results in migration of a small number of low activity flows with minimized packet ordering. The load balancing scheme of LAPS is based on [37] but we minimize the overhead of per flow statistics by using a low cost aggressive flow detector. Furthermore, this scheme does not consider I-Cache locality. whereas LAPS is a more complete solution which maximizes throughput by considering instruction and data cache localities and minimizes packet reorder. Shi et al. also propose an adaptive hashing scheme [22], which assures that the weights of the hashing scheme are modified such that the assignment of flow bundles to cores is more evenly balanced for biased hash bundles found in Internet traffic. In [36], Shi and Kencl propose to combine the previous two schemes i.e., adaptive hashing is used in conjunction with the migration of aggressive bundles. This scheme is complementary to LAPS and can easily be integrated with LAPS to improve the performance of hashing. Guo et al. [16] propose a batch scheduling for packets in order to minimize out of order delivery. However, their experiments assume each packet requires the same application and does not consider I-cache or flow locality in algorithm. Furthermore, their scheme requires expensive synchronization among multiple cores. Some researchers propose the order restoration instead of the order preservation [35]. They allow the packets to be processed out of order on different cores, but only before the packet leaves the system, they are reordered to restore the flow order. Yet, this scheme can have considerable storage overheads, and even worse, packets of the same flow can be processed on different cores, destroying flow locality. Wolf et al. [38] attempt to address the issue of I-cache locality. When a core becomes idle, it searches for a packet of the same application as the previous one. This searching has a lot of overhead and is not feasible for data plane packets. This scheme, although considers application locality, does not consider data locality and packet order.

Some consider a packet processing application as a graph where different tasks within application forms the nodes of the graph [26], [39]. These schemes consider adjacency between nodes for task scheduling as the packet moves between different cores during processing. In contrast, we consider each service as one entity i.e., a packet is tied to a core for the whole processing and do not consider graph scheduling.

VII. Conclusions

We present the design and evaluation of a scheduler for data plane packets in network processor. The packet scheduler adopts an efficient dynamic core allocation scheme for multiple services to improve throughput and to minimize out of order delivery of packets. A key to reducing the out of order packets is to eliminate unnecessary flow migrations. The scheduler achieves this goal by identifying and migrating only the aggressive flows. We present the design of a novel Aggressive Flow Detector (AFD) based on two level caching scheme which integrates readily with our scheduler, and also, is very effective in identifying top aggressive flows with high accuracy. Furthermore, the scheduler extends the hash based design for multi-service routers where the cores are dynamically allocated to services to improve I-Cache locality. Our

experiments with real network traces show that our proposed scheduler improves the throughput by 60% while reducing the out of order packets by 80% as compared to previous schemes.

REFERENCES

[1] Cyclic redundancy code generator. www.actel.com.

[2] The Freescale P4240 processor. http://www.freescale.com.

[3] The University of Auckland traces. http://wand.net.nz/wits/auck/2/.

[4] XLP832 multicore processor. http://www.broadcom.com.

[5] Intel IXP hardware reference manual, January 2003.

[6] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development*, 47(2.3):177 –193, march 2003.

[7] J. D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proceedings of the 14th USENIX conference on System administration*, LISA '00, pages 139–146, Berkeley, CA, USA, 2000. USENIX Association.

[8] Z. Cao, Z. Wang, and E. Zegura. Performance of hashing-based schemes for internet load balancing. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 332–341 vol.1.

[9] G. Chuvpilo, D. Wentzlaff, and S. Amarasinghe. Gigabit ip routing on raw. In *In Proceedings of the 8th International Symposium on High-Performance Computer Architecture, Workshop on Network Processors*, 2002.

[10] K. Claffy, D. Andersen, and P. Hick. The CAIDA anonymized 2011 internet traces.

[11] G. Dittmann and A. Kerkersdorf. Network processor load balancing for high speed links. In *In Proceeding of International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[12] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.*, 32(4):323–336, Aug. 2002.

[13] W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Global Telecommunications Conference, 1999. GLOBECOM '99*, volume 3, pages 1859–1868 vol.3.

[14] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers Boston, MA, 2000.

[15] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[16] J. Guo, J. Yao, and L. Bhuyan. An efficient packet scheduling algorithm in network processors. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 807 – 818 vol. 2, march 2005.

[17] L. Guo and I. Matta. The war between mice and elephants. ICNP, 2001.

[18] F. Hao, M. Kodialam, and T. V. Lakshman. Accel-rate: a faster mechanism for memory efficient per-flow traffic estimation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 155–166, New York, NY, USA, 2004. ACM.

[19] X. Huang and T. Wolf. Evaluating dynamic task mapping in network processor runtime systems. *Parallel and Distributed Systems, IEEE Transactions on*, 19(8):1086–1098, Aug.

[20] M. F. Iqbal and L. K. John. Efficient traffic aware power management in multicore communications processors. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 123–134, New York, NY, USA, 2012. ACM.

[21] L. John and A. Subramanian. Design and performance evaluation of a cache assist to implement selective caching. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 510 –518, oct 1997.

[22] L. Kencl. *Load Sharing for Multiprocessor Network Nodes*. PhD thesis, EPFL, 2003.

[23] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin. A case for run-time adaptation in packet processing systems. *SIGCOMM Comput. Commun. Rev.*, 34:107–112, January 2004.

[24] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21 – 29, march-april 2005.

[25] G. Koren and A. Rosen. Architecture of a 100-gbps network processor for next generation video networks. In *Electrical and Electronics Engineers in Israel (IEEEI), 2010 IEEE 26th Convention of*, pages 000286 –000290, nov. 2010.

[26] J. Kuang and L. Bhuyan. Lata: a latency and throughput-aware packet processing system. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 36–41, New York, NY, USA, 2010. ACM.

[27] Y. Lu and B. Prabhakar. Robust counting via counter braids: An error-resilient network measurement architecture. In *INFOCOM 2009, IEEE*, pages 522–530, April.

[28] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. Elephanttrap: A low cost device for identifying large flows. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 99–108, Aug.

[29] Y. Luo, J. Yu, J. Yang, and L. N. Bhuyan. Conserving network processor power consumption by exploiting traffic variability. *ACM Trans. Archit. Code Optim.*, 4(1), Mar. 2007.

[30] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. CAN, 2005.

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro*, 28(1):69–79, 2008.

[32] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf. An application-aware load balancing strategy for network processors. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 156–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '97, pages 139–152, New York, NY, USA, 1997. ACM.

[34] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown. Analysis of a statistics counter architecture. In *Hot Interconnects 9, 2001.*, pages 107–111.

[35] L. Shi, Y. Zhang, J. Yu, B. Xu, B. Liu, and J. Li. On the extreme parallelism inside next-generation network processors. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1379 –1387, may 2007.

[36] W. Shi and L. Kencl. Sequence-preserving adaptive load balancers. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 143 –152, dec. 2006.

[37] W. Shi, M. MacGregor, and P. Gburzynski. Load balancing for parallel forwarding. *Networking, IEEE/ACM Transactions on*, 13(4):790 – 801, aug. 2005.

[38] T. Wolf and M. A. Franklin. Locality-aware predictive scheduling of network processors. In *In Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS*, pages 152–159, 2001.

[39] Q. Wu and T. Wolf. On runtime management in multi-core packet processing systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 69–78, New York, NY, USA, 2008. ACM.

[40] M. Zadnik and M. Canini. Evolution of cache replacement policies to track heavy-hitter flows. In *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, pages 1–2, Oct.

[41] M. Zadnik, M. Canini, A. Moore, D. Miller, and W. Li. Tracking elephant flows in internet backbone traffic with an fpga-based cache. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 640–644, 31 2009-Sept. 2.