Dynamic Core Allocation and Packet Scheduling in Multicore Network Processors

Muhammad Faisal Iqbal, Jim Holt, Jee Ho Ryoo, Gustavo de Veciana, and Lizy K. John

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 applications 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 my 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. Our light weight hardware implementation shows improvement of 60 percent in the number of packets dropped and 80 percent in the number of out-of-order packet deliveries over previously proposed techniques.

Index Terms—Network processor, load balancing, resource management

1 Introduction

A *Network Processor* is a special-purpose, programmable device that is optimized for network operations. A network processor is generally a multicore processor that can process network packets at wire-speeds of multi-Gbps. Network processors are employed in many demanding network processing environments like core and edge routers. While the main requirement in core routers is high capacity to handle huge amounts of traffic, edge routers require programmability and flexibility in order to support multiple complex applications like intrusion detection, firewalls, protocol gateways, etc. A network processor provides the performance of custom silicon and programming flexibility of general purpose cores. The ability of a network processor to perform complex and flexible processing and its programmability make it an excellent solution for core and edge routers.

A number of network processors exist in the market. These processors can be classified into two categories. The first category includes general purpose multicore processors that are adapted to perform networking functions. Examples of such processors are the ThunderX [1], Sun Niagara [2] and Tilera [3] processors. The second category includes processors which are specifically designed for networking applications. These processors are equipped with hardware accelerators and co-processors in addition to a large number of general-purpose cores. Examples include the Freescale T4240 [4], Broadcom XLP [5], EZChip [6], Cisco nPowerX1 [7] and IBM PowerNP [8]. Both of these categories have a common attribute: they utilize a large number of cores to achieve desirable performance by exploiting parallelism. Networking applications have abundant parallelism because multiple packets can be processed by different cores in parallel. This packet level parallelism makes multicore architectures well suited for networking applications [9]. Network processors with 64 cores or more have been announced by vendors to handle 100 Gbps network speed [10], [11]. With increasing traffic rates and processing demands, the number and complexity of cores in these processors are on the rise and efficiently managing these cores has become very challenging. In this work we focus on dynamic adaptations based on run time traffic behavior in order to optimize performance and make following contributions.

First, design of a hash based packet scheduler and load balancer is presented in order to achieve the goals of preserving flow locality and packet order. A hash based packet scheduler performs very well in order to achieve these goals because it schedules packet at the flow level and thereby maintains packet order and flow locality inherently. A serious impediment to performance of hash based scheduler is the presence of skewed flow sizes in network traffic. Such skewed distribution of flow sizes can result in overloading some cores and may result in packet loss. To avoid packet loss, a load balancer is designed that migrates some flows from the overloaded cores to under-utilized cores. Flow migrations are undesirable because they result in bad data locality and can result in out of order packets. The load balancer proposed in this study minimizes the number of flow migrations by restricting migrations only to the aggressive flows. We present a low cost hardware to identify aggressive flows. Second, the design of the scheduler is extended to support multiple applications in a router where cores can be dynamically allocated to applications. Furthermore, use of incremental hashing is proposed which is low cost

- *M.F. Iqbal, J.H. Ryoo, G.d. Veciana, and L.K. John are with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712-1084.*
  E-mail: {faisaliqbal, jr45842}@utexas.edu, {gustavo, ljohn}@ece.utexas.edu.
- *J. Holt is with Freescale Semiconductor Inc. & MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139.*
  E-mail: jholt@csail.mit.edu, zjim.holt@freescale.com.

Manuscript received 20 Aug. 2015; revised 27 Feb. 2016; accepted 29 Mar. 2016. Date of publication 28 Apr. 2016; date of current version 14 Nov. 2016.
Recommended for acceptance by N. Bagherzadeh.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2560838

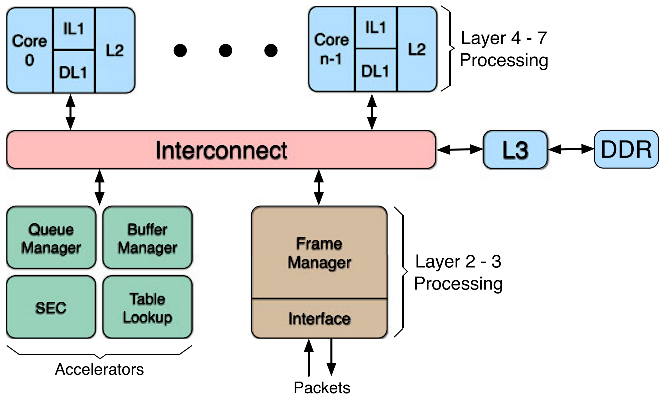


Fig. 1. Typical architecture of a network processor.

method that minimizes number of flow migrations when cores are allocated or deallocated to the services.

In the next section we present architecture of a network processor and discuss issues related to packet scheduling in these processors. The design of the packet scheduler is presented in Sections 3 and 3.3.

2 Background and Motivation

2.1 Architecture of a Network Processor

Many architectural variations of network processors exist in the market. Although vendors differ in specific implementations, they generally share three main features: 1) Multiple cores to exploit packet level parallelism, 2) Accelerators for networking functions and 3) Optimized path for movement of packet data.

Architecture for a typical network processor is shown in Fig. 1. An incoming packet is received by a 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 queue to a processing core. The general purpose core processes the packet and can offload some of the work to accelerators, e.g., it can put some of the work in the queue for security accelerator (SEC). SEC performs the required processing and puts the packet back to the return queue. Eventually, the general purpose core sends the packet back to the FM via an enqueue after finishing the processing.

Network processing can 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. Control Plane processing may involve executing routing protocols like RIP, OSPF, and BGP, or control and signalling protocols such as RSVP or LDP. *Data Plane* deals with actual processing involved in packet forwarding. The data plane execution involves compression, encryption, address searches, address prefix matching for forwarding, classification, traffic shaping, network address translation and so on. In many network processors, the general purpose cores are responsible for processing both data and control plane packets. However, in majority of modern high speed network processors, control plane processing is separated from data plane processing [4], [12]. When a packet arrives, a packet classifier in the FM decides whether it is a control or a data plane packet. Control plane packets take the *slow path* through general purpose cores. The data plane packets (Layer 2 or possibly Layer 3) take the fast path and are not offloaded to general puspose cores. Fast path processing is handled by the FM itself.

The FM is equipped with a large number (32-120) of small cores called I/O Processors (IOP). These IOPs are in-order, dual issue cores with non coherent memory, and generally do not have an operating system. When a packet arrives the packet classifier first identifies whether it is a control plane or a data plane packet. If it is a L2 or possibly L3 data plane packet, it is handled in the FM autonomously by IOPs, otherwise it goes to general purpose cores. This configuration describes a notional system that represents a class of chips as they look today and moving into the next 3-5 years. In this work we are interested in scheduling of data plane packets on IOPs. Since these packets arrive at a very high rate (100 Gbps and even higher in future), an efficient scheduling of packets on IOPs is needed in order to gain good performance. The term IOP and core are used interchangeably in this work.

2.2 Challenges in Packet Scheduling

The design of scheduler for these applications is very challenging. First, the scheduler is in the data path and therefore it should be as efficient as possible. Second, it should meet the requirements of packet ordering, flow locality and cache locality.

2.2.1 Packet Ordering

Although the internet is designed to tolerate out-of-order packets, performance of upper layer protocols, such as Transmission Control Protocol (TCP), greatly depends on packet ordering. Out of order packets can falsely trigger congestion control mechanisms and degrade throughput unnecessarily [13]. Also, applications like Voice Over IP (VOIP) and multimedia transcoding require 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 the packets of a flow. In this work, a flow is defined as a set of packets that have the same source address, destination address, source port, destination port and protocol. If packets from the same flow are processed by different cores, they can 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.

2.2.2 Load Balancing

Load balancing is an important technique to efficiently utilize multiple cores in a network processor. Packets arriving at the input should be distributed uniformly to the available processing cores to maximize performance. An unbalanced allocation of load can swamp some cores. As a result, incoming packets assigned to overloaded cores will experience large delays and may even result in packet loss due to limited storage in the network processor.

2.2.3 Data Cache Locality

If different cores process packets of the same flow, the data cache will be used inefficiently as the same data is copied to multiple caches. Packet processing needs to access per flow data (state, statistics), as well as more global data (routing table). If packets of a flow always go to the same core, locality can be preserved for 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 that are kept per flow, per port etc. Each packet may need to update several of these statistics. If multiple cores work on packets of the same flow in parallel, the 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 degrades performance. The scheduler needs to account for flow locality to achieve good performance.

2.2.4 Support for Multiple Services

Modern network processors are required to support a rich set of services. For example, a multi-service edge router may need to support encryption, decryption, firewalling, intrusion detection and many other services [14], [15], [16]. The packet processing cores used in these processors are usually small with a small instruction cache (i-cache) of size 8-16 KB. These caches can only hold 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. In order to preserve i-cache locality, an efficient resource allocator is needed to divide the pool of processing cores among multiple services. If cores are allocated to services statically at design time based on their worst case requirements, it will result in unnecessary over-provisioning with high system cost. The resource allocator needs to be able to dynamically multiplex cores among services based on runtime traffic requirements in order to keep the processor provisioning level reasonable.

In the next section, we present a packet scheduler which balances load among multiple cores while minimizing out of order packets. In Section 3.3, we extend the load balancer to support multiple services.

3 Load Balancing While Minimizing Packet Reordering

In a network processor, load distribution among the multiple cores is performed by a packet scheduler. A packet scheduler receives an incoming packet form high speed link with traffic rate λ and forwards it to one of the cores for processing. Each cores processing power is μ_i and the total processing power of the network processor is $\mu = \sum \mu_i$. This work focuses on the scheduler for data plane packets and does not consider control plane packets. The design of scheduler for data plane is particularly 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, packet ordering, data cache and instruction cache locality.[1] Previous researchers have presented many load-balancing schemes [17], [18], [19]. These schemes can be classified into two categories:

- *Packet level load balancing:* These schemes schedule each packet independently to achieve uniformity in load assignment. For example packets may be distributed in a round robin fashion [17], or an incoming packet is allocated to the least loaded core [18]. These schemes have two drawbacks. First, these schemes reorder packets very frequently. Second, these schemes cannot utilize the data cache efficiently because they send packets belonging to the same flow to different cores.
- *Flow level load balancing:* Flow level schemes generally use hashing to distribute flows to individual cores [20], [21], [22], [23]. 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 because header fields are constant for all packets of a flow. Hence the flow locality and packet order is maintained.

The scheduler presented in this section uses a hash-based approach because of its simplicity and obvious advantage of packet ordering and flow locality. This section discusses the challenges associated with hash based packet scheduler and presents design of a scheduler that overcomes these challenges.

3.1 Challenges in Hash Schedulers

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. But, there are several dynamic properties of network traffic that make load balancing task challenging for hash-based designs.

3.1.1 Skewed Flow-Bundle Sizes

The quality of hash function plays an important role in distributing the flows evenly to all processing cores. All the flows that map to the same core or bin in the map table are referred to as a flow-bundle. Uniformity of flow bundle sizes means that the hash function has distributed flows very effectively to the processing cores. Under ideal conditions, each flow bundles should have a size of $\frac{F}{M}$. Where F is the number of active flows and M is the number of bins or cores.

Many researchers have worked on designing effective hash functions for internet addresses [22], [24], [25]. It has been shown that CRC16 performs very well for internet traffic [24]. Similar results are observed in this study.

3.1.2 Skewed Flow Sizes

Even with perfect distribution of flows to cores, load imbalance can still occur because all the flows are not of the same size. In fact, it is well known that network traffic constitutes only few heavy-hitter (high data rate) flows and many low data rate flows [23], [26]. Fig. 2 demonstrates this behavior in real network traffic. The plot shows the popularity of flows (y-axis) with most popular flow plotted first (x-axis).

Shi et al. have shown that hashing alone cannot balance the load under this highly skewed distribution of flow sizes [23]

1. This section does not consider i-cache locality. The scheduler presented in this section is extended in section 3.3 to make it i-cache aware for multi-service routers.

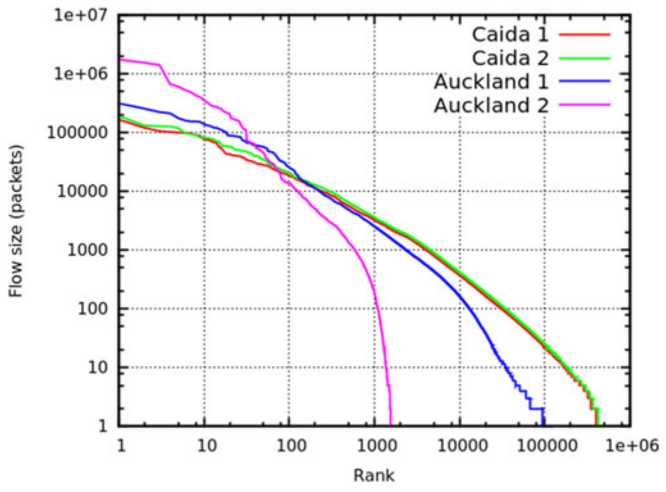


Fig. 2. Distribution of flow sizes in real network traces. Rank 1 is the flow with the highest flow size.

and can result in overloading some cores. In this scenario, the load on each core should be monitored and adjusted dynamically to migrate some load to underutilized cores. Care must be taken because it is desirable to minimize the number of flow migrations. Flow migrations result in out-of-order packets and also badly affect data cache performance.

In order to minimize the number of flow migrations, previous research has made the observation that migrations should be limited to only top aggressive flows [23]. In this way load balance can be achieved by minimum number of flow migrations. However, previous research based its study purely on offline analysis and kept per flow statistics to identify aggressive flows. Maintaining per flow statistics has a lot of overhead and is not possible in realistic designs. Although many per flow statistics are maintained by software, accessing those software statistics is very time consuming for a scheduler that is trying to schedule data plane packets. The data plane packet scheduler needs to function with minimum software intervention for good performance.

This research presents the design of a hardware scheduler for data plane packets. A novel low-overhead hardware technique to identify aggressive flows is presented. The aggressive flow detection scheme is based on two-level caching idea of annex-cache [27] used in general purpose applications. The caching based aggressive flow detector integrates readily with a hash based packet scheduler. The complete design and evaluation of the scheduler is presented in this section.

3.2 Packet Scheduler Design

The proposed packet scheduler uses a hash based design which is a natural way of maintaining flow locality and packet order. The scheduler is called Locality Aware Packet Scheduler (LAPS). 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. The map table[2] stores target core ID where the packet is eventually forwarded. In the presence of skewed flow size distribution as shown in Fig. 2, the scheduler identifies and migrates the aggressive flows from the overloaded core to achieve load balance. An efficient scheme for identifying and migrating aggressive flows is presented.

2. Map table is used instead of direct hashing because it allows dynamic core allocation presented in the next section.

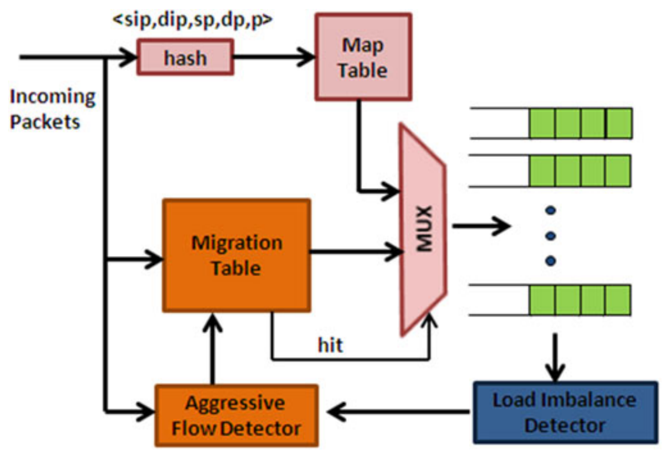


Fig. 3. Load balancer design.

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 that are waiting in the overloaded core's queue.

To avoid the above two situations, it is desirable to minimize the number of flow migrations. If only the most aggressive flows can be identified and migrated, load balance can be achieved with minimum disruption, i.e., only a few flows need to be migrated to achieve load balance. In order to achieve this, a low cost mechanism is need to identify top aggressive flows. This research proposes a novel cache based hardware called Aggressive Flow Detector (AFD) to identify the top flows. The hardware consists of a 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 5.3.

Fig. 3 presents the scheduler design. The incoming packets are hashed to get index into a map table that stores the target core IDs. On load imbalance, the incoming packet flow to the overloaded core is migrated to the least loaded core if the flow is identified as an aggressive flow by AFD. The decision is recorded in the Migration table. So the future packets of the same flow are always migrated to the newly allocated core. The scheduler gives priority to the output of migration table over the default hash table. If the input queue indicated by the scheduler is filled up, the incoming packet is dropped.

3.2.1 *Aggressive Flow Detection*

The design of Aggressive Flow Detector is based on *annex cache*. Annex Cache was proposed by John [27] to exploit locality in the memory references in general purpose

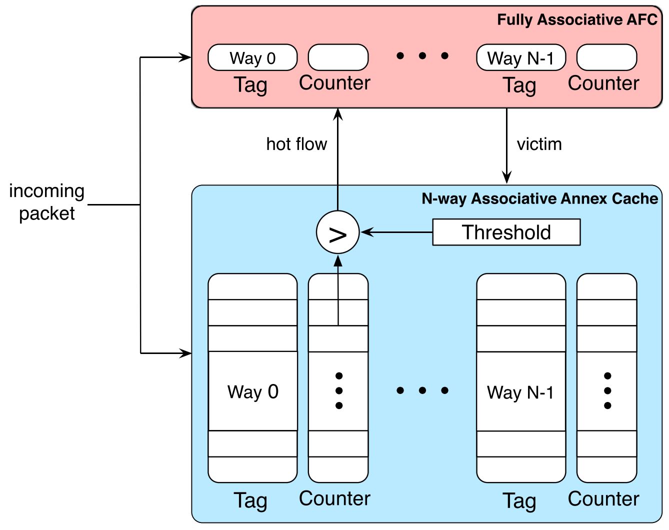


Fig. 4. Structure of aggressive flow detector.

processor workloads. This study shows that such a structure can be very useful in to identify aggressive flows.

The AFD has two main components as shown in Fig. 4. One component is a small fully associative cache called Aggressive Flow Cache. 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 [27] because in AFD annex cache is bigger than AFC. A larger AFC is undesirable because the proposed scheme wants to limit the number of monitored aggressive flows. Annex cache is a bigger structure that serves as a qualifying station for large number of flows to demonstrate 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 pre-defined threshold. The threshold for promotion to AFC is the LFU count in AFC. If the hit count in annex cache 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.

3.2.2 Load Imbalance Detection

Length of the longest queue is used to detect the load imbalance in the system, i.e., when the length of the longest queue in the system reaches a predefined threshold, load imbalance signal is asserted. As long as the load imbalance signal is asserted, all the aggressive flows are migrated to the least loaded core. The migrated flows are forwarded to the new core even after the load imbalance signal is de-asserted as a result of flow migration.

Most modern network processors have dedicated hardware units for management of packet queues [4], [8], [28] and a lot of research has been done on design of these hardware queue managers [29], [30], [31]. These queue managers implement different active queue management algorithms (e.g., Random Early Detection RED) and monitor the queue length as part of their normal operation. This queue length information can easily be used by the load balancer to detect the need for low migration, i.e., it can easily be reported to the packet scheduler when the queue reaches a threshold. Hence, additional hardware resources are not needed to monitor queue length, because queues are already monitored for congestion control purposes. In this work, it is assumed that hardware queue manager monitors the queue state and generates the load imbalance signal.

3.3 Packet Scheduler for Multiservice Routers

A simple hash based design as presented in 3 can result in inefficient i-cache usage. In order to exploit i-cache locality, LAPS divides the pool of cores among all active applications or services. In effect, there is a separate map table for each service. All cores in a single map table always get packets that require same processing. Hence, i-cache locality is preserved. The main question is how to allocate cores to applications? If cores are allocated on compile time, we need provisioning for worst case traffic requirements of each service requiring a huge number of cores. Fortunately, all services do not experience their worst case traffic simultaneously and hence cores can be multiplexed dynamically among services to keep the total number of cores reasonable. Many dynamic allocation schemes have been proposed in the past. In this work we adapt the policies presented in [32] to integrate it with hash load balancer. We further make it flow aware so that the number of flow migrations are minimized on dynamic allocation and deallocation of cores. LAPS utilizes incremental hashing (also known as Linear hashing) to minimize number of flow migrations on dynamic adaptation.

3.3.1 Allocation of Cores to Services

LAPS keeps a list of cores that are marked as surplus cores by other services (Section 3.3.2). When a service becomes overloaded, LAPS looks through the list of surplus cores and finds the core that 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.

3.3.2 Release of Cores by Services

When input queue to a core becomes empty, a timer starts. When the timer reaches idleth, 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 (before the core is put to deep sleep state by the power management scheme), 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

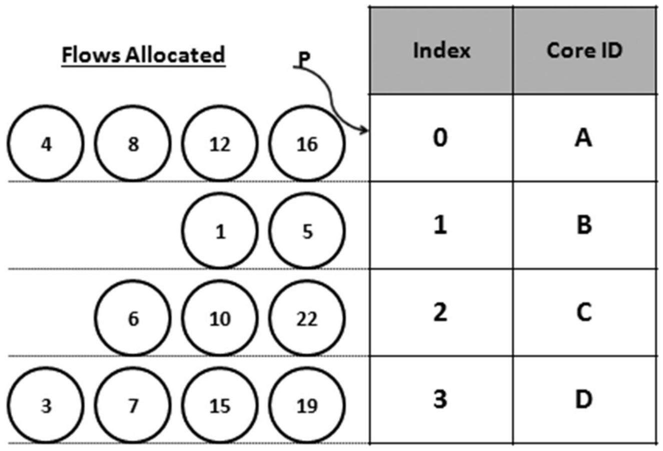


Fig. 5. Initial assignment of flows. $m = 4$, $p = 0$, $h_0(k) = k\%4$..

the performance overhead is tolerable because this service is only lightly loaded anyway. The value of idleth is set to $10\,us$ based on previous research [33].

3.3.3 Load Redistribution on Core Allocation

When an additional core is allocated to a service, the resource manager appends the core ID to the end of the hash table for that service, i.e., hash table size grows by 1. *Linear Hashing* scheme allows a hash table to grow one bucket at a time and does not require rehashing of all flows currently allocated. This makes it useful for load balancing because it is desirable to minimize the flow disruption when an additional core is allocated to a service. The Linear Hashing scheme was introduced by Litwin [34] and has been described in [35]. Following is a brief introduction of how this scheme works.

3.3.4 Initial Assignment of Flows

The linear hashing scheme has m initial buckets labelled 0 through $m-1$, and an initial hashing function $h_0(k) = f(k)\%m$ that is used to map any key k to one of the m buckets, and a pointer p that points to the bucket to be split whenever new bucket is added. Initial value of p is 0. An example is shown in Fig. 5. In this example, $h_0(k) = k\%m$ is used as a hash function for simplicity.

3.3.5 Bucket Split

When the resource manager allocates an addition core to a service, bucket 0, that is pointed by p, is split into two buckets: the original bucket 0 and a new bucket m. The flows originally mapped to bucket 0 by hash function h_0 are now distributed between bucket 0 and m using a new hash function h_1. Fig. 6, shows layout of linear hashing after the new core bucket has been added to the map table. The shaded flows are the flows that are moved to the new bucket. Bucket 0 has been split and and the flows originally in bucket 0 are distributed between bucket 0 and bucket 4, using a new hash function $h_1(k) = k\%8$. When another additional core is allocated, i.e., another bucket m+1 is added to the hash table, the flows mapped to bucket 1 will now be redistributed using $h1$ between buckets 1 and m+1. A crucial property of $h1$ is that the keys that were mapped to some bucket j by h_0, are

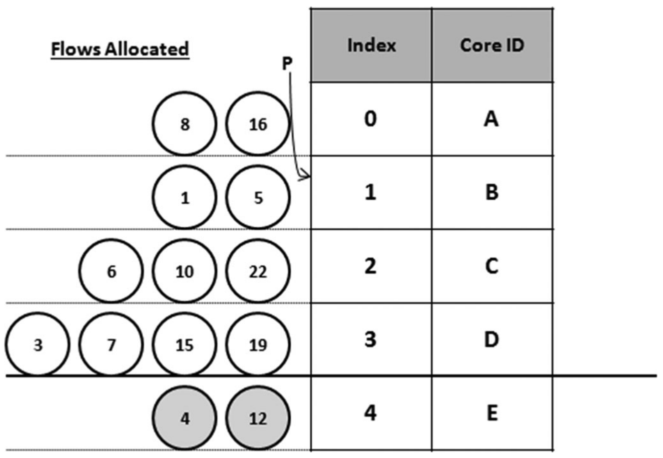


Fig. 6. Flow redistribution after allocation of an additional core. $p = 1$, $h_0(k) = k\%4$, $h_1(k) = k\%8$.

remapped to either j or bucket $j + m$. This is a necessary property for linear hashing to work. An example of such hashing function is: $h_1(k) = k\%2m$.

3.3.6 Round and Hash Function Advancement

After enough core allocations, all original m buckets will be split. This marks the end of splitting round 0. During round 0, p went from 0 to $m-1$. At the end of round 0, there are $2m$ buckets in the hash table. Hash function h_0 is no longer needed because all $2m$ buckets can be addressed by h_1. Variable p is reset to 0, and a new round namely round 1 starts. A new hash function h_2 needs to be used. Fig. 7 shows the state of hash table at the end of splitting round 0. In general, the linear hashing scheme uses a family of hash functions h_0, h_1, h_2, and so on. Let the initial function be $h_0(k) = f(k)\%m$, then any later hash function is $h_i(k) = f(k)\%2^i m$. This way it is guaranteed that if h_i hashes a key to the bucket $j \in [0..2^i m - 1]$, h_{i+1} will hash the same key to either j or bucket $j + 2^i m$. At any time, two hash functions h_i and h_{i+1} are used. In general, in splitting round i, hash functions h_i and h_{i+1} are used. At the beginning of round i, $p = 0$ and there are $2^i m$ buckets. When all those buckets are split, splitting round $i+1$ starts, p goes back to zero, the number of buckets become $2^{i+1} m$, and hash functions h_{i+1} and h_{i+2} will start to be used.

3.3.7 Summary and Mapping Scheme

Initially, each service is allocated m cores, i.e., there are m buckets in the hash table. At any time the hash table manager has the following components:

1) A variable i that indicates the current splitting round.
2) A variable p that points to the bucket to be split next.
3) A total number of $2^i m + p$ buckets in the hash table.
4) Two hash functions h_i and h_{i+1}. The base hash function used is CRC16, i.e., $f(k) = CRC16(k)$.

Whenever a packet arrives, the hash scheduler has to map it to one of the buckets in the map table. The mapping scheme works as follows:

$$h(k) = \begin{cases} h_{i+1}(k) & : h_i(k) < p \\ h_i(k) & : h_i(k) \geq p. \end{cases}$$

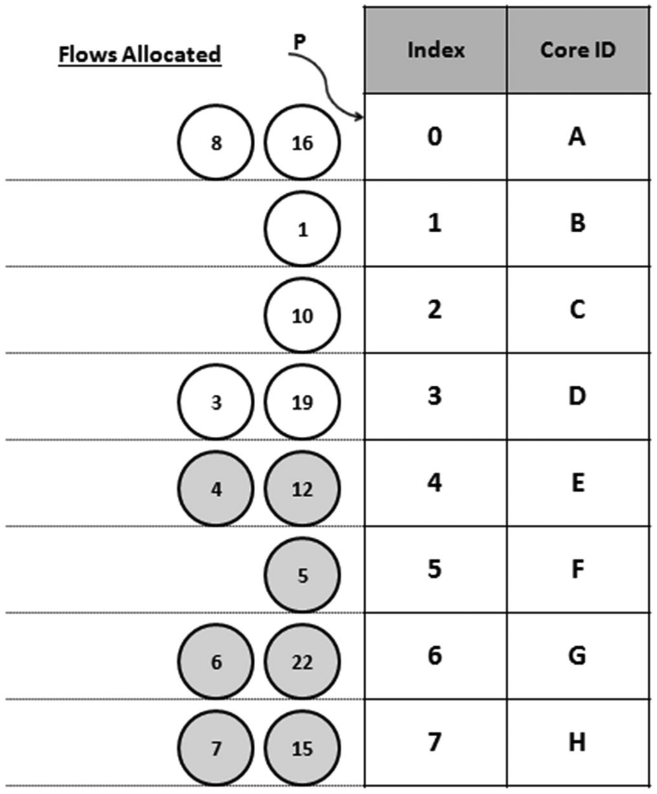


Fig. 7. Flow redistribution at the end of round 0 (beginning of round1). $p = 0$, $h_1(k) = k\%8$, $h_2(k) = k\%16$.

That is, if $h_i(k) \geq p$, choose bucket $h_i(k)$ because this bucket has not been split yet in the current round. If $h_i(k) < p$, choose bucket $h_{i+1}(k)$. The value of p is incremented whenever a new core is allocated to the service. Use of this incremental hashing in conjunction with load balancing scheme of Section 3.2 allows us to add additional cores to a service with minimal disruption to the existing flows.

3.3.8 Load Redistribution on Core Release

When a core is reallocated to another service, it is removed from the bucket list of the victim service. Essentially, a process that is reverse of load redistribution on allocation takes place. The value of round i is updated, i.e., $i = (b/m) - 1$, where b is the current number of buckets in the map table. The value of p is set to $b - 2^i m$. and the hash function is also changed accordingly.

3.4 Overall Scheme

Fig. 8 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.

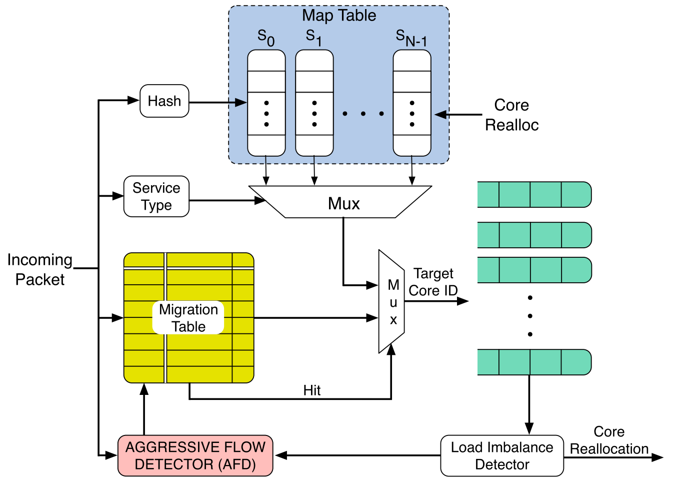


Fig. 8. Locality aware packet scheduler.

3) Under load imbalance, the aggressive flows (flows that hit in AFC) are migrated to the least loaded core allocated to that service similar to the load balancing scheme of Section 3.
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.

4 Evaluation Infrastructure

4.1 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.

4.1.1 CAIDA Traces

This dataset contains anonymized traffic traces from CAIDA's equinix-sanjose monitor [36]. 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.

4.1.2 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 [37]. All connections from the university to external world pass through this measurement point. These traces are of one hour long duration each.

4.2 Workload Model

In order to model the different services running on a multiservice router we consider a workload similar to the one presented in Fig. 9. This model is based on methodology presented in [16]. 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 services represented by different paths of Fig. 9. *Path 1* describes the path of outgoing packets which are tunnelled

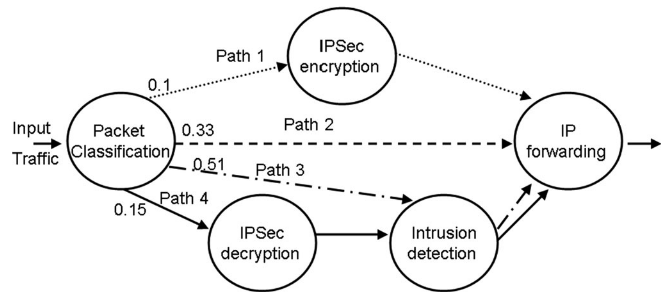


Fig. 9. Example task graph for an edge router.

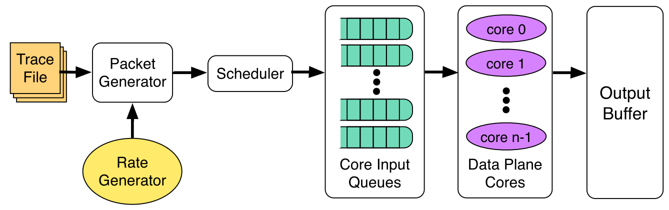


Fig. 10. Simulation infrastructure.

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.

4.3 Simulation Infrastructure

For evaluating different scheduling strategies, we developed a simulation model in SpecC [38]. SpecC is similar to systemC [39] in its design and philosophy. Different components of the simulator are shown in Fig. 10.

4.3.1 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 [40]. The traffic rate is governed by the equation (1).

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

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. 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.

4.3.2 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 pervious research [19]. A packet is lost when it is assigned to a queue which is already full.

TABLE 1
List of CAIDA Traces Used in the Study

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 2
List of Auckland-II Traces Used in the Study

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 3
Data Plane Core Configuration

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

4.3.3 Processing Latencies

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

$$PD_i = T_{proc,i} + FM_{penalty} + CC_{penalty,} \tag{2}$$

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 [41] simulator. The configuration of in-order cores is shown in Table 3.

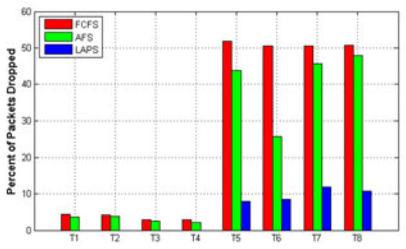
We executed these packet processing applications and derived a packet processing delay model for each service. T_{Proc} is measured to be 0.5μs for path 2, i.e., IP forwarding. For path 3 it is measured to be 3.53μs. For Path 1 it also depends on the packet size and is given as

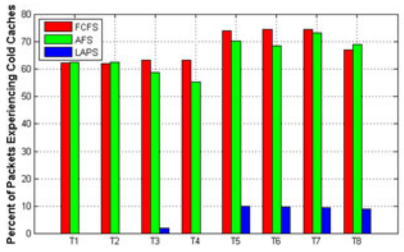
$$T_{proc,path1} = 3.7\mu s + \frac{PacketSize}{64\ byte} \times 0.23\ \mu s. \tag{3}$$

Similarly the processing time for path 4 is given as

$$T_{proc,path4} = 5.8\mu s + \frac{PacketSize}{64\ byte} \times 0.21\ \mu s. \tag{4}$$

$FM_{penalty}$ is set to four cache misses (0.8μs) 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. Becasue of small I-cache, these cores can hold instructions of only the last executed program (e.g., AES encryption used in IPSec


(a) Packets Dropped


(b) Percentage of Cold Caches

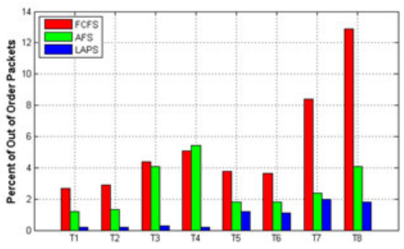

(c) Out of order packets

Fig. 11. Comparison of LAP with FCFS and AFS with different traffic scenarios listed in Table 6.

requires 16 KB). So whenever a packet of different service arrives at a core, it will experience cold cache penalty. We set the cold cache penalty to 10 μs which is the cold cache penalty for the smallest service, i.e., IP Forwarding as observed in GEMS simulator. 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 the D-cache misses due to context switch in this work.

5 Results and Discussion

5.1 Throughput Improvement with LAPS

LAPS aims to improve throughput by exploiting locality in instruction and data caches. Fig. 12 shows effectiveness of LAPS in improving throughput of a sixteen core system. In this experiment, all four services of Fig. 9 are active. Simulation infrastructure of Fig. 10 is used. The traffic rate generator is configured to increase the traffic gradually to measure the maximum throughput supported by the system. Traffic is equally divided among the four services, i.e., Path 1 through 4 of Fig. 9. Caida 1, Caida 2, Caida 3 and Caida 4 traces are used for generating packets for Path 1, Path 2, Path 3 and Path 4. Fig. 12 compares throughput of LAPS with a First Come First Served (FCFS) and Arbitrary Flow Shift (AFS) scheduler. X-axis shows combined input traffic rate which is equally divided among all the services and y-axis shows the traffic rate observed at the output.

FCFS scheduler services packets in their arrival order and does not consider flow or instruction locality. As a result, it causes many data and instruction cache misses and results in the worst throughput among the three schedulers. As compared to FCFS, AFS reduces some flow migrations and is able to improve throughput a little. But AFS is still unaware of instruction locality and results in suboptimal performance. In comparison to these two schemes, LAPS improves both flow and instruction locality and results in substantially better throughput (56 percent more than AFS and about 100 percent more than FCFS). Ideal throughput represents a system with no cache miss penalties. The plot is obtained by setting the cache miss penalties to zero. Although such a system is infeasible, it represents a theoretical maximum which can be achieved if the system has full knowledge of everything and is able to move data and instructions into caches before they are needed.

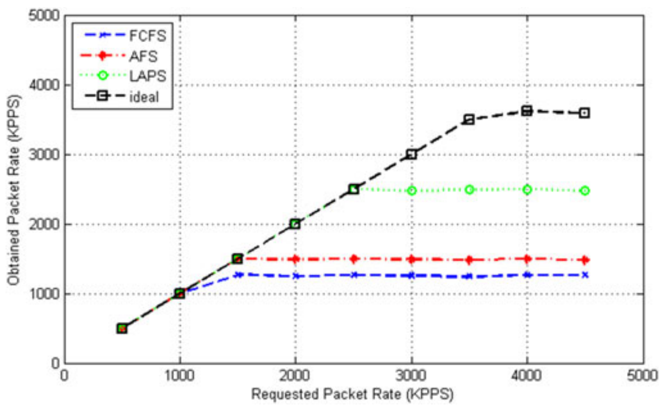


Fig. 12. Throughput comparison of different schedulers.

Note that the throughput supported by the simulated sixteen core system is much less than the industrial system. There are two reasons for this: First, the software implementations of services are taken from open source benchmark suites where as companies use highly optimized implementations. Second, the experiments are based on software only implementations and do not use hardware accelerations.

5.2 Performance Improvement with LAPS

In this section, we compare the performance of LAPS with a First Come First Served scheduler and the scheduler presented in [20]. This scheduler migrates arbitrary flows when load imbalance is detected. We call this scheme Arbitray Flow Shift. 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 4. Set 1 represent 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.

For each service, we use real network traces to generate the packet. We used different sets of traces listed in Table 5.

TABLE 4
Parameters Governing Traffic Rate

	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

Rate is in Mpps and period is in seconds

TABLE 5
Traces Used in Experiment for Packets of Individual Services

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

The combination of sets of equation in Table 4 and traces in Table 5 creates difference traffic scenarios listed in Table 6.

Fig. 11a shows packets dropped with three schemes under the traffic scenarios shown in Table 6. LAPS outperforms FCFS and AFS in both underload and overload conditions. FCFS and AFS dsitrubute packets of different services arbitrarily to cores and suffer from poor I-cache locality (Fig. 11b). These schemes drop packets even in underload conditions because almost 60 percent 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 inputr rates. Fig. 11c shows the effectiveness of LAPS in preserving packet order under traffic scenarios of Table 6. FCFS does not care for packet ordering and hence results in most out of order packets. AFS reduces these out of order packets but still there are considerable amount of out of order packets due to large number of flow migrations. LAPS minimizes the flow migrations by only migrating the top flows and hence result in very few packets being delivered out of order. Next, we show how our proposed Aggressive Flow Detector helps in identifying the top flows and helps to achieve load balance with minimum flow migrations.

5.3 Performance of Aggressive Flow Detector

The proposed AFD has two components: An aggressive flow cache, and an annex cache. An annex cache can be viewed as a preliminary filter where non-aggressive flows are filtered out from entering the small AFC. Therefore, any entry in AFC is a 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 the 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 offline analysis is considered a false positive. Fig. 13a 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 a 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 percent accuracy with a 512 entry annex cache. Caida traces have much more flows active and thus require a larger annex cache. In Caida 1 and 2 respectively, only 14 and 13 most aggressive flows are correctly

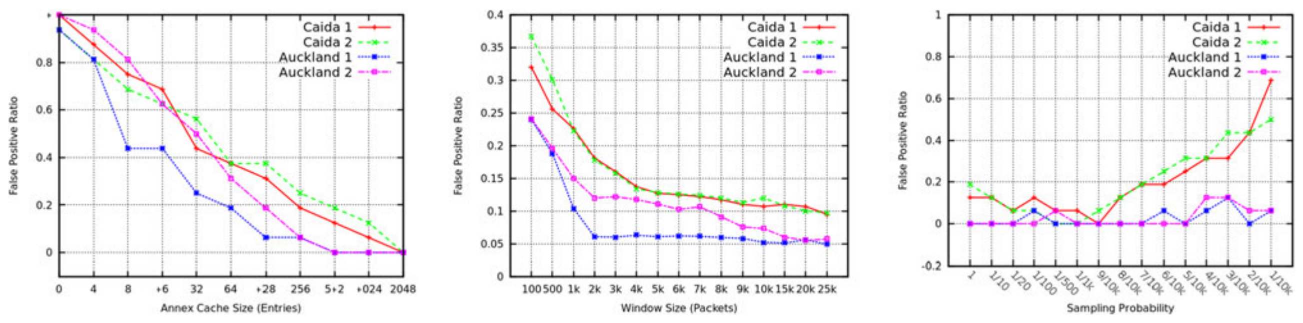
TABLE 6
Traffic Scenarios Used in Fig. 11

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

identified with a 512 entry annex cache. When we double the size to 1,024 entries, accuracy improved an average of 6.25 percent. 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 intervals. In Fig. 13b, we performed the same accuracy evaluation with varying interval steps. In this experiment, we assumed the fixed 512 entries for the size of the annex cache. Our mechanism shows above 90 percent accuracy from a small step size such as every 1,000 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. Fig. 13c shows the false positive ratio when packets are sampled with a probability p and not all packets 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 have generally 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.

5.4 Dynamic Behavior of the System

In order to observe the effectiveness of of dynamic resource allocation scheme, temporal behavior of number of cores allocated to each service is plotted. Fig. 14 shows the dynamic behavior when two services are active in the system. Service 1 is the same as Path 1 of Fig. 9, i.e., the outgoing VPN traffic is encrypted using IPSEC encryption. Service 2 is the Path 3 of Fig. 9 which corresponds to processing incoming packets through a firewall. The traffic requirements of each service are varied over time and the response of the resource allocation system is observed. Fig. 14 shows that the system is very effective in following the changing traffic requirements and

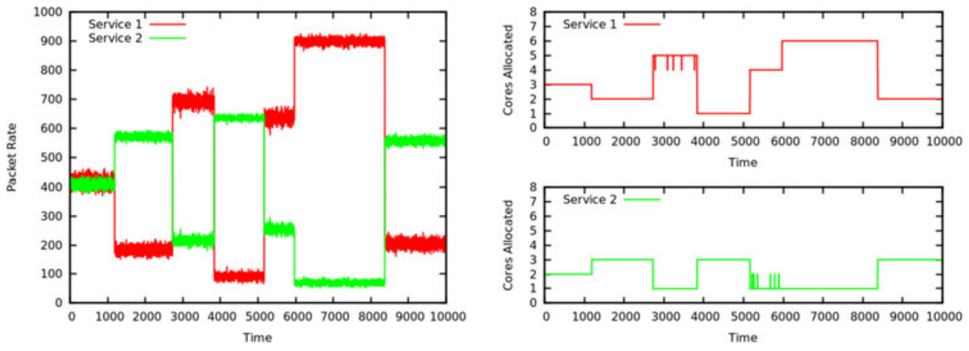


(a) False Positive Ratio in a 16 entry AFC when Annex Cache size is varied

(b) Effect of window size on accuracy of AFD

(c) Effect of packet sampling on performance of AFD

Fig. 13. Effectiveness of AFD in identifying aggressive flows.



(a) Traffic variation over time

(b) Number of cores allocated over time

Fig. 14. Temporal behavior of the resource allocator.

changes the core allocations to match the demands of each service very effectively.

5.5 Opportunities of Migration without Reordering

The results presented in the previous section show significant improvements in minimizing number of flow migrations and percentage of packets which leave the system out of order. However, there is still a small percentage of packets which are out of order (1-2 percent). Ideally, any out of order packets should be eliminated. In this section we study, the opportunities of migrating flows without any risk of packet reorder. Such opportunities exist because the gap between the packets may be large enough to allow flows to migrate without any risk of reordering [42].

To investigate such opportunities a simple experiment is conducted. Whenever a packet arrives, the opportunity count is incrementd by 1 if the target queue does not contain any packets from the same flow. Table 7 shows the opportunity counts for Caida 1 trace. Note that this is a conservative estimate because even if packets exist in the queue, it is still possible to migrate without reordering if the existing packets will be processed before the new incoming packet.

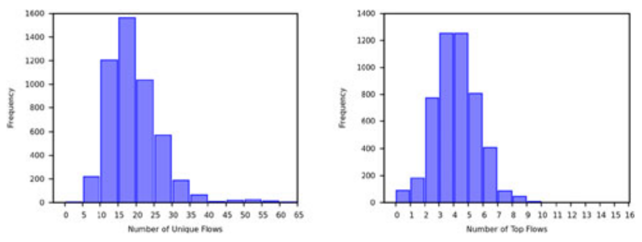
TABLE 7
Opportunities For Flow Migration without Reordering

Flow Type	Qlength	Total Packets	PMO
Any	Any	28219211	20385555
Any	$> qthresh$	14068978	9375921
Top	$> qthresh$	619555	252825

First row shows, when any packet arrives and it is safe to migrate it without any reordering. The second row shows the same results when the input queue length of the target core is greater than the imbalance threshold. This is more important than because this is the situation where flow migration is needed. The third row represents the situation when the incoming packet belongs to the top flow and the target core is overloaded. This is the situation where our scheme can benefit by migrating the large flow without causing any reordering. This is a great opportunity to further minimize out of order departure of packets. It is possible that current scheme when moves a big flow does not cause reorder. But the scheme in itself does not have any such mechanism. It can be further extended such that when its time to migrate flow, priority could be given to flows with minimum packets in the system to minimize reordering. This will help minimize reorderings but will make the system little more complex. Because now, we need to store the target cores for the large flows and also need to keep track of the number of packets belonging to that flow in the system. Such a system is part of future work.

5.6 Analysis of Flows on Migration

In order to understand the behavior of the system, the flows allocated to the overloaded core at the instance of flow migration are analyzed. Fig. 15a shows the number of unique flows present in the queue of the overloaded core when a big flow is migrated from that core. Generally, a large number (15-20) of flows are present. This indicates that the overload is caused by a combination of large and small flows and migrating the large flow is expected to



(a) Number of flows allocated to overloaded core at the time of migration (b) Number of big flows allocated to overloaded core at the time of migration

Fig. 15. Analysis of flows allocated to overloaded core at the time of migration.

mitigate the load imbalance. A small number of flows would indicate that the overload is caused by small number of large flows and there is a potential that migrating the large flow would result in imbalance even in the newly allocated core. Fig. 15b shows the number of big flows allocated to the overloaded core at the time of flow migration. From the plot it can be seen that the imbalance is usually caused by multiple big flows and migrating one flow is not expected to cause imbalance in the new core.

Presence of multiple big flows in the queue of overloaded core opens up the opportunity to further improve the scheme. For example, when migration decision is made preference could be given to the big flow which will cause less distortion in the order of packets, e.g., the flow with less number of packets in the queue could be preferred over the flow with larger number of packets. Such a scheme is likely to increase the complexity of the system becasue now core association of the flows and their number of packets in the system need to be monitored. Design of such a system is part of future work.

6 Related Work

6.1 Existing Work on Packet Ordering

Previous researchers have adopted two different approaches to minimize packet reordering in network processors: order restoration and order preservation.

6.1.1 *Order Restoration*

This technique allows packets belonging to a flow to be processed out of order by different cores and restores the order at the output [8], [18], [19]. At the input, each packet is tagged with a sequence number and the packets are allowed to exit the system in strict sequence order. Per flow tagging is needed in order to preserve order among packets of the same flow. This requires keeping per flow information, which is a huge overhead as there can be millions of flows active at a time [43], [44]. Overhead of per flow tagging can be reduced by using global tagging. Global tagging is easy to implement but it is overly restrictive as it forces order even among packets of different flows and results in throughput degradation. Order restoration also requires an expensive synchronization mechanism because multiple cores may be required to update the ordered list of packets of the same flow at the same time. This scheme also results in poor data cache locality because flow locality is not preserved.

6.1.2 *Order Preservation*

This technique avoids the overheads of order restoration by preserving packet order. One example of order preservation is the batch scheduling scheme presented by Guo et al. [17]. In this scheme, a batch of packets is dispatched to cores in a strict round robin manner and is read from cores in the same order. This scheme does not require per flow information but requires expensive synchronization among multiple cores and is not suitable to be implemented for data plane packets. Furthermore, this scheme assumes that each packet requires the same application and does not consider i-cache or flow locality in the algorithm. Another method to preserve packet order is to use a hash function to distribute packets to processing cores [20], [21], [22], [23]. This work adopts hash based scheme and integrates a hash scheduler with a dynamic resource allocator.

6.2 Existing Work on Load Balancing

Dittman presented a hash based packet scheduler and load balancer [20]. When a load imbalance is detected, this scheme migrates arbitrary flows to an under-loaded core. Such a scheme blindly migrates flows and can result in a large number of flow migrations. This scheme is referred to as arbitrary flow ship in this paper. A large number of flow migrations results in poor data cache locality and causes many out of order packets. Shi et al. [23] proposed to only migrate the flows that have high data rates. The load balancing scheme presented in this work is based on [23] but the proposed scheme minimizes the overhead of per flow statistics by using a low cost aggressive flow detector. Furthermore, the load-balancer presented in [23] does not consider i-cache locality whereas this research presents a more complete solution that maximizes throughput by considering instruction and data cache localities and minimizes packet reordering. Shi et al. also proposed an adaptive hashing scheme [21] that 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 [22], 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 the solution proposed in this research and can easily be integrated with the proposed scheduler to further improve the performance of hashing.

The load balancer proposed in this research limits the flow migration only to the aggressive flows. In order to achieve that, efficient identification of aggressive flows is required. Detecting and monitoring aggressive flows is an important part of traffic management and policing. Consequently, there has a plethora of work on how to calculate flow statistics. Initial naive proposals to keep counters for each flow [45], [46] are not scalable when there are 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 [43], [47], [48], [49], [50] to find the accurate estimate of the rates of aggressive flows. In contrast, the proposed packet scheduler in this research merely needs to identify the top aggressive flows without accurately estimating the rates of all flows. The closest related work is done by Yi et al. [44] where a single cache is used to identify "elephant" flows. Experiments done in this research reveal that a single level caching scheme can result in large number of false positives due to many "mice" flows active at any time. This research proposes a novel two-level caching scheme to identify aggressive flows based on annex cache [27]. The proposed detector effectively eliminates the false positives and integrates directly with the scheduler.

6.3 Existing Work on Dynamic Resource Allocation

Many researches have observed the need for dynamic resource allocation in network processors [14], [51], [52] and there have been proposals for runtime resource allocations in the past [15], [53], but these schemes consider a packet processing application as a graph where different tasks within the application form the nodes of the graph. These schemes consider adjacency between nodes for task scheduling as packets move between different cores in a pipelined manner during processing. In contrast, this research considers each service as a single entity, i.e., a packet is tied to a single core for the whole processing and graph or pipeline scheduling is not considered. Wolf et al. [54] observed that 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. This results in huge performance overhead. They attempted to address the issue of i-cache locality through careful packet scheduling. When a core becomes idle, their scheme 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. Although this scheme considers application locality, does not consider data locality and packet order. This work presents a more complete solution to the problem of packet scheduling and resource allocation in multi-service routers in contrast to the prior proposals which have focused on the individual aspects of the problem. A preliminary version of this paper has appeared as a conference paper [55].

7 Conclusion

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 aggressive flows. We present the design of a novel Aggressive Flow Detector 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 percent while reducing the out of order packets by 80 percent when compared to previous schemes. The schemes presented in this paper show promising improvements over the previous work. Hash based designs of packet scheduler and resource manager have very low overhead. This makes the designs very scalable for data rates of 100 Gbps and even beyond.

References

[1] L. Gwennap, “Thunderx rattles server market,” *Microprocessor Rep.*, Jun. 2014, http://linleygroup.com/mpr/article.php?id=11223.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded SPARC processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar./Apr. 2005.

[3] G. Chuvpilo, D. Wentzlaff, and S. Amarasinghe, “Gigabit IP routing on raw,” in *Proc. 8th Int. Symp. High-Perform. Comput. Archit., Workshop Netw. Process.*, 2002, pp. 2–9.

[4] The Freescale P4240 processor (2012) [Online]. Available: http://www.freescale.com

[5] T. R. Halfhill. (Oct. 2012). Broadcom samples 28nm XLP ii [Online]. Available: http://www.linleygroup.com/newsletters/newsletter_detail.php?num=4901&year=2012&tag=3

[6] EZCHIP, “NPS-400 Network Processor,” 2013, http://www.tilera.com/products/?ezchip=598&spage=603.

[7] CISCO, “Next-generation custom routing silicon processors for the internet of everything,” 2013, http://www.cisco.com/c/en/us/solutions/collateral/optical-networking/ons-15454-series-multiservice-provisioning-platforms/solution_overview_c22-729610.pdf.

[8] 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 J. Res. Develop.*, vol. 47, pp. 177–193, Mar. 2003.

[9] P. Duggisetty, “Design and implementation of a high performance network processor with dynamic workload management,” Master’s thesis, Univ. Massachusetts Amherst, Amherst, MA, USA, Sep. 2015.

[10] Tilera72 core network processor tile-gx72, (2013). [Online]. Available: http://www.tilera.com

[11] Broadcom 64 core processor iBCM-88030, (2012). [Online]. Available: http://www.dcom.com/press/release.php?id=s666869

[12] B. Wheeler. (Sep. 2013). A new era of network processing [Onlien]. Available: http://www.linleygroup.com/cms_builder/uploads/ericsson_npu_white_paper.pdf

[13] V. Paxson, “End-to-end internet packet dynamics,” in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 1997, pp. 139–152.

[14] 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.*, vol. 34, pp. 107–112, Jan. 2004.

[15] Q. Wu and T. Wolf, “On runtime management in multi-core packet processing systems,” in *Proc. 4th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2008, pp. 69–78.

[16] X. Huang and T. Wolf, “Evaluating dynamic task mapping in network processor runtime systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 8, pp. 1086–1098, Aug. 2008.

[17] J. Guo, J. Yao, and L. Bhuyan, “An efficient packet scheduling algorithm in network processors,” in *Proc. 24th Annu. Joint Conf. IEEE Comput. Commun. Soc.*, Mar. 2005, pp. 807–818.

[18] L. Shi, Y. Zhang, J. Yu, B. Xu, B. Liu, and J. Li, “On the extreme parallelism inside next-generation network processors,” in *Proc. 26th IEEE Int. Conf. Comput. Commun.*, May 2007, pp. 1379–1387.

[19] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf, “An application-aware load balancing strategy for network processors,” in *Proc. 5th Int. Conf. High Perform. Embedded Archit. Compilers*, 2010, pp. 156–170.

[20] G. Dittmann and A. Kerkersdorf, “Network processor load balancing for high speed links,” in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst.*, 2002, pp. 727–735.

[21] L. Kencl, ”Load sharing multiprocessor network nodes,” PhD dissertation, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2003.

[22] W. Shi and L. Kencl, “Sequence-preserving adaptive load balancers,” in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, Dec. 2006, pp. 143–152.

[23] W. Shi, M. MacGregor, and P. Gburzynski, “Load balancing for parallel forwarding,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 8, pp. 790–801, Aug. 2005.

[24] Z. Cao, Z. Wang, and E. Zegura, “Performance of hashing-based schemes for internet load balancing,” in *Proc. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc*, 2000, pp. 332–341.

[25] X. Hesselbach, R. Fabregat, B. Baran, Y. Donoso, F. Solano, and M. Huerta, "Hashing based traffic partitioning in a multicast-multipath MPLS network model," in *Proc. 3rd Int. IFIP/ACM Latin Amer. Conf. Netw.*, 2005, pp. 65–71.
[26] L. Guo and I. Matta, "The war between mice and elephants," in *Proc. 9th Int. Conf. Netw. Protocols*, 2001, pp. 180–188
[27] L. John and A. Subramanian, "Design and performance evaluation a cache assist to implement selective caching," in *Proc. IEEE Int. Conf. Comput. Des. VLSI Comput. Process.*, Oct. 1997, pp. 510–518.
[28] G. Koren and A. Rosen, "Architecture of a 100-gbps network processor for next generation video networks," in *Proc. IEEE 26th Conv. Elect. Electron. Eng. Israel*, Nov. 2010, pp. 286–290.
[29] I. Papaefstathiou, T. Orphanoudakis, G. Kornaros, C. Kachris, I. Mavroidis, and A. Nikologiannis, "Queue management in network processors," in *Proc. Int. Conf. Des., Autom. Test Eur.*, 2005, pp. 112–117.
[30] W. Zhou, C. Lin, Y. Li, and Z. Tan, "Queue management for QoS provision build on network processor," in *Proc. 9th IEEE Workshop Future Trends Distrib. Comput. Syst.*, 2003, pp. 219–224.
[31] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, and A. Herkersdorf, "Accelerating packet buffering and administration in network processors," in *Proc. Int. Symp. Integr. Circuits*, 2007, pp. 373–377.
[32] A. Raghunath, A. Kunze, E. J. Johnson, and V. Balakrishnan, "Framework for supporting multi-service edge packet processing on network processors," in *Proc. ACM Symp. Archit. Netw. Commun. Syst.*, 2005, pp. 163–171.
[33] M. F. Iqbal and L. K. John, "Efficient traffic aware power management in multicore communications processors," in *Proc. 8th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2012, pp. 123–134.
[34] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Proc. 6th Int. Conf. Very Large Data Bases*, 1980, pp. 212–223.
[35] E. Horowitz, S. Shani, and D. Mehta, *Fundamentals Data Structures in C++*. 2 ed., Silicon Pr, Summit, NJ, USA, 2006.
[36] K. Claffy, D. Andersen, and P. Hick. The CAIDA anonymized 2011 internet traces, (2011). [Online]. Available: http://www.caida.org/data/passive/passive_2011_dataset.xml
[37] The University of Auckland traces, (2000). [Online]. Available: http://wand.net.nz/wits/auck/2/
[38] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Boston, MA, USA: Kluwer, 2000.
[39] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer, 2002.
[40] J. D. Brutlag, "Aberrant behavior detection in time series for network monitoring," in *Proc. 14th USENIX Conf. Syst. Admin.*, 2000, pp. 139–146.
[41] 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," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, Nov. 2005.
[42] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 51–62, Mar. 2007.
[43] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. IEEE Int. Conf. Comput. Commun.*, Apr. 2009, pp. 522–530.
[44] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "Elephanttrap: A low cost device for identifying large flows," in *Proc. 15th Annu. IEEE Symp. High-Perform. Interconnects*, Aug. 2007, pp. 99–108.
[45] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown, "Analysis of a statistics counter architecture," in *Proc. 9th Symp. High Perform. Interconnects*, 2001, Art. no. 107-.
[46] W. Fang and L. Peterson, "Inter-as traffic patterns and their implications," in *Proc. Global Telecommun. Conf.*, vol. 3, pp. 1859–1868, 1999.
[47] F. Hao, M. Kodialam, and T. V. Lakshman, "ACCEL-RATE: A faster mechanism for memory efficient per-flow traffic estimation," in *Proc. Joint Int. Conf. Meas. Modeling Comput. Syst.*, 2004, pp. 155–166.
[48] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, pp. 270–313, Aug. 2003.
[49] M. Zadnik, M. Canini, A. Moore, D. Miller, and W. Li, "Tracking elephant flows in internet backbone traffic with an FPGA-based cache," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Sep. 2009, pp. 640–644.
[50] M. Zadnik and M. Canini, "Evolution of cache replacement policies to track heavy-hitter flows," in *Proc. 6th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2010, pp. 31:1–31:2.
[51] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur, "Multiprocessor scheduling in processor-based router platforms: Issues and ideas," in *Proc. 2nd Workshop Netw. Process.*, 2003, pp. 48–62.
[52] A. Satheesh, D. Kumar, and S. Krishnaveni, "Dynamic adaptive self-configurable network processor," in *Proc. Symp. Workshops Ubiquitous, Autonomic Trusted Comput.*, 2010, pp. 160–164.
[53] J. Kuang and L. Bhuyan, "Lata: A latency and throughput-aware packet processing system," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 36–41.
[54] T. Wolf and M. A. Franklin, "Locality-aware predictive scheduling of network processors," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2001, pp. 152–159.
[55] M. F. Iqbal, J. Holt, J. H. Ryoo, G. de Veciana, and L. K. John, "Flow migration on multicore network processors: Load balancing while minimizing packet reordering," in *Proc. 42nd Int. Conf. Parallel Process.*, 2013, pp. 150–159.

Muhammad Faisal Iqbal did his BS in Electronic Engineering from GIK Institute, TOPI, Pakistan in 2003. He received his MS and PhD degrees from the University of Texas at Austin in 2009 and 2013 respectively. His interests include microprocessor architecture, power and performance modeling, workload characterization and low power architecture.

Jim Holt is a Senior Software Engineer at Intel. Previously, He led the Processor Architecture and Modeling Team for Freescaleâs Networking and Multimedia group, and was also a Visiting Scientist at the Massachusetts Institute of Technology where he was a Principal Investigator for the Angstrom project which investigated self-aware 1000-core chips of the future. He has 32 years of industry experience focused on microprocessors, multicore systems, software engineering, distributed systems, design verification, and design optimization. He is an IEEE Senior Member, and has over 30 refereed publications. He earned a PhD in Electrical and Computer Engineering from the University of Texas at Austin, and an MS in Computer Science from Texas State University-San Marcos.

Jee Ho Ryoo is a PhD student at the University fo Texas at Austin under the supervision of Professor Lizy K. John. He received the BS degree in electrical and computer engineering from Cornell University in 2011 and the MS degree in electrical and computer engineering from the University of Texas at Austin in 2014. His research interests include computer architecture, memory systems and performance analysis. He is a member of the IEEE and the IEEE Computer Society.

Gustavo de Veciana (S'88-M'94-SM'01-F'09) received his BS, MS, and PhD in electrical engineering from the University of California at Berkeley in 1987, 1990, and 1993 respectively, and joined the Department of Electrical and Computer Engineering where he is currently a Cullen Trust Professor of Engineering. He served as the Director and Associate Director of the Wireless Networking and Communications Group (WNCG) at the University of Texas at Austin, from 2003-2007. His research focuses on the analysis and design of communication and computing networks; data-driven decision-making in man-machine systems, and applied probability and queueing theory. Dr. de Veciana served as editor and is currently serving as editor-at-large for the IEEE/ACM Transactions on Networking. He was the recipient of a National Science Foundation CAREER Award 1996 and a co-recipient of five best paper awards including: IEEE William McCalla Best ICCAD Paper Award for 2000, Best Paper in ACM TODAES Jan 2002-2004, Best Paper in ITC 2010, Best Paper in ACM MSWIM 2010, and Best Paper IEEE INFOCOM 2014. In 2009 he was designated IEEE Fellow for his contributions to the analysis and design of communication networks. He currently serves on the board of trustees of IMDEA Networks Madrid.

Lizy Kurian John is B. N. Gafford Professor in the Electrical and Computer Engineering at UT Austin. She received her PhD in Computer Engineering from the Pennsylvania State University. Her research interests include workload characterization, performance evaluation, architectures with emerging memory technologies, and high performance processor architectures for emerging workloads. She is recipient of many awards including the NSF CAREER award, UT Austin Engineering Foundation Faculty Award, Halliburton, Brown and Root Engineering Foundation Young Faculty Award 2001, University of Texas Alumni Association (Texas Exes) Teaching Award 2004, The Pennsylvania State University Outstanding Engineering Alumnus 2011, etc. She has coauthored a book on Digital Systems Design using VHDL (Cengage Publishers, 2007), a book on Digital Systems Design using Verilog (Cengage Publishers, 2014) and has edited 4 books including a book on Computer Performance Evaluation and Benchmarking. She holds 9 US patents. She is an IEEE Fellow (Class of 2009).