Experience with High-Speed Automated Application-Identification for Network-Management[*]

Marco Canini
DIST, University of Genoa

Wei Li
Computer Lab., Cambridge Univ.

Martin Zadnik
CESNET, CZ

Andrew W. Moore
Computer Lab., Cambridge Univ.

ABSTRACT

AtoZ, an automatic traffic organizer, provides control of how network-resources are used by applications. It does this by combining the high-speed packet processing of the NetFPGA with an efficient method for application-behavior labeling. AtoZ can control network resources by prohibiting certain applications and controlling the resources available to others. We discuss deployment experience and use real traffic to illustrate how such an architecture enables several distinct features: high accuracy, high throughput, minimal delay, and efficient packet labeling — all in a low-cost, robust configuration that works alongside the enterprise access-router.

1. INTRODUCTION

The evolution of the Internet has encouraged new data-intensive applications to emerge and gain quickly in popularity. Examples include Peer-to-Peer (P2P) file-sharing, online video services (e.g., YouTube), IPTV and wide-area file storage. Meanwhile, many business critical or highly interactive applications such as VoIP, financial trading platforms and real-time multi-player games are delay and loss sensitive, and thus prone to starvation due-to other data-intensive applications. Simple over-provisioning is insufficient and comes with its own drawbacks.

There have been two main kinds of approaches in application-based traffic-shaping systems. The first is based upon a list of known IP-addresses and port-services, such as the Linux netfilter, and a second approach is based upon individually-identified flows and through the use, for example, of deep-packet inspection (DPI) modules, which is common as part of commercial products (e.g., Qosmos QWork, Blue Coat PacketShaper, and Riverbed Steelhead) and NIDSes (e.g., Snort [1], Bro [2]).

In contrast we present AtoZ, an automatic traffic organizer that allows flexible traffic management on a per-application basis. We envisage use in an enterprise environment: that is, a single link between organization (or office-location) and Internet. Our prototype deployments are in-line with the access router in the same manner as NIDSes (Network Intrusion Detection Systems). Our approach is a novel packet-labeling system combining a machine-learning software system and a high-speed hardware subsystem leading to a novel solution that meets all our design objectives — high throughput, high accuracy, swift packet-labeling, minimum delay and zero (unintended) packet loss.

AtoZ is built upon a hybrid hardware-software platform and combines two significant elements: a novel hierarchical packet labeling scheme based on accurate behavioral application identification, and a high performance data path implemented in hardware using the NetFPGA platform[3].

Our contribution is a flexible architecture that has substantial advantages over operating system-based solutions or existing commercial application-traffic shaping solutions. With lower computational and memory requirements, AtoZ handles multiple full-duplex gigabit line-rates and provides accurate and efficient labeling of packets, with minimal delay and zero unintended packet loss.

We have conducted evaluations using real network traffic from a university campus and a research institution to show how specialized hardware for packet-processing allows for high accuracy application-labeling with minimal impact on performance.

We envisage AtoZ deployment on the granularity of *access-policy*: an enterprise, department, or hall-of-residence. Using this approach the advantages of deployment are enjoyed by those bearing its cost.

2. TRAFFIC CLASSIFICATION

Recent research in the area of traffic classification has brought many interesting methods that can identify applications without relying on "well-known" port numbers, or looking at the contents of packet payloads. Most of these newer schemes classify traffic by recognizing statistical patterns in externally observable characteristics (e.g., [4, 5]. For a given flow identified by the {src IP, src port, dst IP, dst port, proto.} 5-tuple, a common approach is to assign the membership of the flow to one of several application groups based just on flow features such as the number and size of packets, inter-packet arrival times, or flow duration. A big difference among these methods is the feature-set.

[5] shows that high-accuracy can be achieved with only the first few packets of each flow. We use such light-weight early-flow identification mechanism to provide accurate and

[*]This work was done while Marco Canini and Martin Zadnik were visiting the Computer Lab., Cambridge University.

timely labeling results, although our approach is not dependent upon this specific method.

Further, we note that usually, for any given traffic, there is only one unique application for a distinct {IP, port, proto.} 3-tuple within a given time period. For example, TCP packets to and from `207.46.107.73:1863` would be associated to MSN Messenger. This observation leads to the (reasonable) assumption that packets associated to the same 3-tuple commonly belong to the same application in a certain time period. This means that *automatically* learned {IP, port, proto.} rules can be used to accurately label packets at a higher level of aggregation than the standard 5-tuple. Thus, significantly reducing the memory cost maintaining a mapping table based on the 5-tuple while also accelerating the labeling process.

We test our assumption on observations of institutional networks with traces taken over multiple-day periods since 2003 until the current day. Within each 24 hour period, we identify flows through pattern matching and manual inspection of the payload contents. The results show that traffic follows our stability assumption within each 24-hour period. The only exceptions that exist are the different applications encapsulated in VPNs, tunnels or SOCKS proxies. In our current implementation these applications are identified as a special class of their own. Other mechanisms can be applied to identify the encapsulated applications as needed [6].

3. DESIGN

Several significant features allow our system to meet our objectives. A hybrid architecture (§ 3.2) allows the fast-path to support high-throughput, minimum packet-delay and no packet-loss, while maintaining functionality (e.g., application identification) in software components. The hierarchical packet-labeling (§ 3.1) is combined with accurate application identification (§ 4.1) to obtain high labeling accuracy while significantly reducing the overall time-to-bind[1] and occupation of the flow cache. Further, the modular structure (§ 3.3) of our system allows flexible updates and extensibility of the system. In the following sections we present the design, discuss the trade-offs and show its advantages over standard approaches.

3.1 Hierarchical Packet Labeling

The packet labeling in our system, illustrated in Figure 1, is hierarchically structured: it comprises of the Host Cache (HC) and Flow Cache (FC). The HC stores entries that contain dynamically learned rules to label the packets associated to hosts and service activities that cause significant traffic on the link. A rule maps an {IP, port, proto.} 3-tuple to an application class. Packets not labeled by the HC are handled by the FC, which implements the mapping between a flow (identified by its 5-tuple) and an application class. Finally, all labeled and unlabeled packets are forwarded for application-specific management.

The application-class labels in both caches are automatically learned. Specifically, the HC is dynamically updated by inferring the binding of an application to a certain 3-tuple: an aggregation of application information derived from flows identified earlier. Once the 3-tuple association has been established, subsequent packets can immediately

[1]We define time-to-bind as the period between the time a new flow is first observed and the time it is identified.

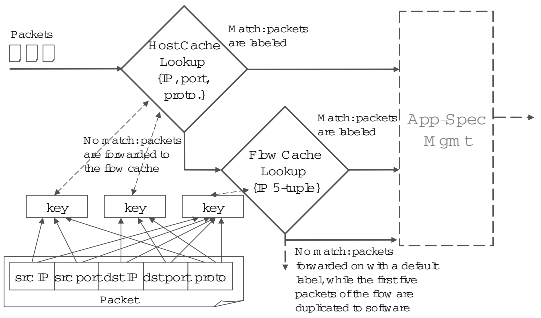
Packets
HostCache Lookup {IP, port, proto.}
Match: packets are labeled
No match: packets are forwarded to the flow cache
Flow Cache Lookup {IP 5-tuple}
Match: packets are labeled
App-Spec Mgmt
key
key
key
src IP
src port
dst IP
dst port
proto
Packet
No match: packets forwarded on with a default label, while the first five packets of the flow are duplicated to software

Figure 1: Hierarchical Packet Labeling

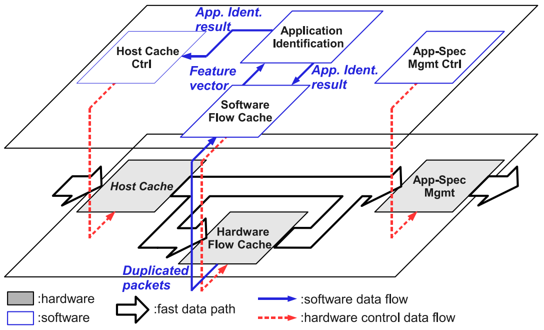

Figure 2: The hybrid architecture and data flow in the traffic organizer.

be labeled at this higher aggregation. This means that *new flows* involving a *known 3-tuple* are handled without requiring to track and identify them; and they have an instantaneous time-to-bind.

In common with past-authors [7], we capitalize on the observation that host interactions on the Internet exhibit a heavy-tail behavior, that is, the majority of the traffic is associated to only a small proportion of the interacting hosts. The HC deals with the most frequently used services: the heaviest traffic load. The HC also reduces the utilization of flow records in the FC, since a significant number of flows may be handled by fewer HC entries.

However, the HC does not label packets carrying 3-tuples for which there is no known association. To guarantee that the rest of traffic is also handled timely and correctly labeled, we complement the HC with the FC which supports the maintenance of per-flow state information. Further, the FC serves to select the subset of initial packets from each flow that are needed to identify applications.

The application-identification module provides flow identification to initiate updates of the FC and HC as further discussed in Section 4.2.

High packet-labeling accuracy is achieved by the resulting hierarchical application-identification mechanism. We remove the time-to-bind for up to 79% of the flows and save up to 75% of the FC occupation in our evaluation.

3.2 Hybrid Architecture

The NetFPGA has meant we are able to implement the data path for packet processing at full-duplex Gbps line rates with minimal processing delay measured in only a few clock cycles [3]. Its capabilities are only limited by the complexity of the logic and the size of available memory; therefore, our system adopts a hybrid architecture that integrates the

NetFPGA as a fast packet processor with a more comprehensive, albeit slower, software data path executed on a host machine to overcome memory and complexity limitations.

Several design challenges exist for such a hybrid architecture. The first challenge is the gap between sophisticated, dynamic organizing-functions and heavily constrained complexity and memory availability in hardware. The second challenge is to be disruption-free: that is, to minimize the impact on performance, at speed, delay and (unintended) packet loss. The third challenge is to find the right balance between the high-speed hardware and the lower speed software. The communication between hardware and software through the PCI bus is limited, operating at rates far slower than the total fast-path throughput. This requires the hardware to realize load shedding for the software, handling the majority of the traffic. One final challenge is that the hardware ought to allow dynamic rule updates, of the FC and the HC, on-the-fly.

These challenges are resolved with appropriately designed data paths and a distribution of processing tasks between hardware and software components, leading to an uninterrupted packet processing data path in hardware while minimizing HBA-host transfers, limiting the complexity of gateware, reducing hardware memory usage, and maintaining a functional modularization of each system component permitting component-mobility between hardware and software.

Core packet-processing components, aka the fast-path: the packet-labeling process based on the HC and FC hierarchy, the Application-specific Management logic, and the packet-receive and transmit units are each implemented in hardware. In testing, our prototype switches traffic to a different physical interfaces. Traffic-shaping extensions (e.g., rate-limiting) for specific applications are also available.

The components not located on the fast data path are implemented in software. These are the software flow cache, the application identification module, and the hardware controllers.

Moreover, the software only needs observe the first few packet headers of a flow for it to derive classification results. And so, the computational cost in this architecture can be significantly reduced by limiting the portion of data to be processed in software. Hence, we design a packet filter in hardware that duplicates only the first few (typically five) packet headers of each unmatched flow to the software.

The hardware and software interface is data-driven, e.g., immediately after identifying a flow, the software forces an update of the corresponding hardware FC entry on-the-fly.

The packet filtering and the de-synchronization between hardware and software guarantee that the processing capability of software components will not be a bottleneck in the system. Consequently, the whole system is able to work at a constant high speed, handling the maximum system specification (8 Mpps) with an average delay of less than 17 μs.

3.3 Modular Structure

Like the Click Modular Router [8], we design our system as a set of functional modules. Each component in our system can be flexibly reorganized and tuned, for (a) extensibility: facilitating further research, (b) flexibility: allowing the system to be re-assembled for specific tasks or different purposes, and (c) adaptability: enabling easy update of identification models or traffic organizing policies to tackle novel development of Internet applications. For example, the application identification module can be replaced with other machine-learned models, DPI methods, or even hybrid classifiers. Further, host-based classification models may be used alongside the flow identification to derive HC rules.

4. SYSTEM COMPONENT DESIGN

The traffic organizer is comprised of several modules, as shown in Figure 2. The modules may combine codes on two different platforms: gateware on the NetFPGA card and software on the host machine. For convenience we refer to them as hardware components and software components.

Hardware components focus on supporting a fast data path providing minimal packet-processing delay and no unintended packet loss. Accordingly, software components are designed for two objectives: firstly, to support sophisticated functions such as rule creation and flow identification, and secondly, to manage and control the hardware components.

In the following subsections, we describe a system design which is based upon several functional modules. We show how the modules connect with each other, and how software and hardware components collaborate in each module to achieve the design goals. Here we focus on the high-level functional design, while hardware design and implementation is discussed in Section 5.

4.1 Application Identification

The hierarchical packet labeling scheme is supported by an application identification module whose goal is to classify each observed flow with the category of the application(-type) to which it belongs. The classification method used in our prototype is summarized in this section, but is not the primary contribution of this work. A thorough description is in [5].

Using flow features collected from the first several packet headers of a set of training flows and a small amount of pre-classified ground-truth data, the method employs a C4.5 supervised machine learning algorithm to build real-time classification models. These models are accurate, but are trained off-line due to the need of ground-truth data.

For this work, we implemented in C++ the flow-feature computation and classification scheme providing a method suitable for online execution with a low overhead (the C4.5 implementation devolves to several nested `if` statements). As in [5], we use a set of 12 behavioral features derived from the first five packets of each flow to classify traffic into 13 application classes that group applications based upon their purpose (e.g., MSN Messenger text-window conversations being classified as “CHAT”).

4.2 Host Cache

As mentioned (§ 3), the Host Cache serves to label packets involving hosts responsible for large amounts of traffic flows.

The HC is a two-stage mechanism, comprising a fast packet labeling stage in hardware, and a management stage in software. The packet labeling stage operates at the per-packet level, matching three packet header fields, either {dst IP, dst port, proto.} or {src IP, src port, proto.} against a set of existing rules based on {IP, port, proto.} 3-tuples. If it finds a match, the packet is labeled using the application class associated with the matching rule; otherwise it retains a default label.

These rules are created and managed by the HC management stage, using results from the application identification module. It operates by defining a time window in terms of seconds during which candidate rules are formed from the aggregation of newly classified flows (5-tuples) into server[2] end-points: {IP, port, proto.} 3-tuples. At the end of each time window, we score each candidate rule using a utility metric. Note that only flows starting after the beginning of the current measurement interval are counted. To each end-point corresponds a flow count x and a set of application classes obtained by application identification. Let p be the proportion of flows belonging to the class with the majority of memberships, then: $U = f(x, p)$ where $f()$ is a function increasing with x and p[3]. The idea is that the utility is proportional to the flow count and is affected by the consistency of the application identification results. Then we select the rules whose utility exceeds a threshold U_{th}. If the rules are more than the HC capacity H, the top-H rules with higher utility are kept.

Lastly, we need to consider the active rules: those already in use in the hardware stage. For active rules, we estimate their utility as $U = n$ where n is the number of flows matched by this rule during the past time window. However, we aim to minimize the use or hardware memory resources, and so in the case of TCP traffic, U is measured by counting the SYN+ACK packets matching a given rule. For UDP traffic, we use a space-efficient, shared Bloom filter [9] to track the number of distinct flows by hashing a key derived from the rule 3-tuple fingerprint and the flow 5-tuple.

Finally, the management stage identifies the new rule set by sorting the candidate together with the active rules. Active rules that are not considered sufficiently useful, or have existed longer than a time-to-live period are evicted and replaced by new candidate rules. When an entry is evicted, then the full classification path is required for the future traffic it corresponds to.

The utility score serves to quantify the contribution of a rule. Using the utility metric allows us to directly trade uncertainty in classification-stability (i.e., the applicability of previous classifications to future traffic) for efficiency gains made through the use of the HC.

4.3 Flow Cache

Flow level operations are supported by a dual flow cache, combining two data structures: hardware Flow Cache (HFC) and software Flow Cache (SFC), located on the NetFPGA and on the host machine, respectively. The role of the dual flow cache is to maintain flow state information for currently active flows (inactive flows are identified using a timeout). Specifically, the HFC only observes and processes packets which are not matched by the HC. Furthermore, the SFC processes only the headers of several packets of each flow, according to the principle that the hardware supports a far larger throughput than software.

Both the HFC and the SFC are bi-directional, i.e., there is only one entry for packets of the same flow in both directions. However, the HFC is devoted to packet labeling, and maintains minimal information about the flows (stored in a space-efficient format). On the other hand, the SFC supports the flow identification by collecting the flow features

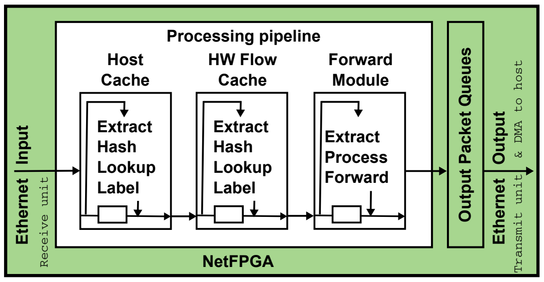

Figure 3: FPGA gateware – Fast processing pipeline and the NetFPGA environment.

from the first several packet headers.

Furthermore, the HFC is managed by the SFC using a special asynchronous and data driven mechanism. For example, the software updates the application class of a certain flow once it is classified and manages the flow timeout[4]. Finally, it removes the HFC entry when a flow expires.

Importantly, stricter hardware constraints require that the HFC has a smaller capacity than the SFC. When changes in the traffic patterns cause an increase in the number of flows leading to a saturation of its capacity (e.g., during DoS attacks), the HFC can only label all packets for a subset of identified flows in the SFC. Counter-measures such as elephant flows detection [10] or reducing the flow timeout can overcome the impact of such an attack with the drawback of (potentially) increased classification inaccuracy. Section 5.2 describes a simple LRU-based eviction policy used to maintain the most active flows in the HFC.

5. HARDWARE DESIGN

While software components provide the control plane of the whole system, the hardware implements the entire data path on a NetFPGA card. The gateware in the NetFPGA is described in Verilog and can be ported to any network interface card equipped with FPGA chips.

5.1 The Fast Packet Data Path

The hardware components form a fast data path consisting of several modules, as shown in Figure 3. Our hardware modules work in cut-through mode: each processes frames (Ethernet encapsulated packets) immediately upon data are available and independently of any other module. Modules receive and transfer data using a 64-bit wide data bus with a FIFO-like interface, and are designed to process up to 8 Mpps which is sufficient to handle four fully utilized Gigabit interfaces irrespective of packet sizes.

The life of a packet in the pipeline begins in the HC. Firstly, the IP addresses, ports and protocol are extracted from the packet. Then two hash values are computed, the first using {src IP, src port, proto.} and the second using {dst IP, dst port, proto.}. These hash values are used to look up the application class label in the HC table. The mechanics of the hash table is discussed in detail in Section 5.2. If the lookup is successful, then the frame header is annotated with the class label. In this case, the frame

[2]The server side is the recipient of TCP connections or the first destination of UDP flows.

[3]The $f()$ in this prototype is the linear function $x \times p$, and further investigation is planned in the future work to also include a packet count per end-point.

[4]The flow timeout is managed in software only to maintain a simple hardware logic. When a flow times out, the software resets the packet filtering counter to test whether the flow is still active. If no packets are observed for this flow during another timeout period, the flow is considered expired.

passes the HFC without any further processing. Otherwise, the HFC extracts the IP 5-tuple and computes a single hash value (since we consider bi-directional flows) which is used to look up the flow record in the HFC table. If found, the flow record contains the class label and a packet counter. If the label is valid, the frame header is annotated with it. The counter is only used to keep track of the first few packets for this flow (typically 5) so that their headers can be duplicated to software. When the counter reaches its maximum, aside from TCP packets with FIN or RST flags set, no more packets for this flow are duplicated to software. However, the software can reset the counter when required (e.g., to handle a flow-timeout). When the lookup fails, a new flow entry is created in the HFC table, with counter of zero and an empty class label. The class label is updated as soon as the new flow is identified.

Both labeled and unlabeled frames are forwarded for application-specific management. In our implementation, this is done by the Forward Module which is capable of forwarding traffic to different output ports and/or to modify destination MAC addresses based on the application class. The implementation of this module can vary depending on the desired functionality. Examples include priority queues to accelerate critical traffic, filters and rate-limiters to throttle undesired traffic, intelligent load balancers or specific modules that apply different Service Level Agreements to different application classes.

5.2 Hash Table Lookup Scheme

The lookup scheme in both the HC and HFC is based on a Naïve Hash Table (NHT). The concept of the NHT is based on using a hash function to divide the search space into disjunctive buckets with approximately the same number of entries in each bucket. Searches are made by computing the hash value of a request descriptor to locate its corresponding bucket, which is then searched for the target entry.

We are aware of other schemes, such as those based on Bloom filters, e.g., [11, 10], but NHT is readily mapped on FPGA on-chip memory structure and inherently has characteristics that other schemes might miss or implement only with some difficulty, NHT allows addition and removal of items quickly and continuously, and the storage and update of per-entry state information.

As both tables are stored in the on-chip memory, space is a major limitation. At the expense of accuracy, we store in each entry only a small fingerprint of the original item instead of its complete descriptor (e.g., 5-tuple). This greatly increases the number of available entries, but it introduces the possibility that two different items (e.g., flows) collide into the same entry if they have the same hash value and fingerprint. This undesired situation is called a false positive. However, by dimensioning the bit-length b of the fingerprint, the probability of false positives p_f can be reduced to an acceptable rate. This probability can be estimated by the birthday problem. For 2^h buckets, the bit-length of the hash value and the fingerprint is $h + b$, therefore we have:

$$p_f = 1 - \frac{m!}{m^n(m-n)!} = 1 - \frac{2^{(h+b)}!}{2^{(h+b)^n}(2^{(h+b)} - n)!}$$

$$\approx 1 - e^{-\frac{(n-1)n}{2\times 2^{(h+b)}}}$$

where m is the total number of hash values and n is the number of entries in use. Table 1 gives the false positive rate for several configurations of the NHT.

Number of entries	Bit-length of hash $(h+b)$		
	40	44	48
4K	$1.9 \cdot 10^{-6}$	$1.2 \cdot 10^{-7}$	$7.5 \cdot 10^{-9}$
8K	$4.1 \cdot 10^{-5}$	$3.2 \cdot 10^{-7}$	$2.4 \cdot 10^{-8}$
16K	$1.2 \cdot 10^{-4}$	$7.6 \cdot 10^{-6}$	$4.8 \cdot 10^{-7}$
32K	$4.9 \cdot 10^{-4}$	$3.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-6}$

Table 1: Probability of false positives (p_f)

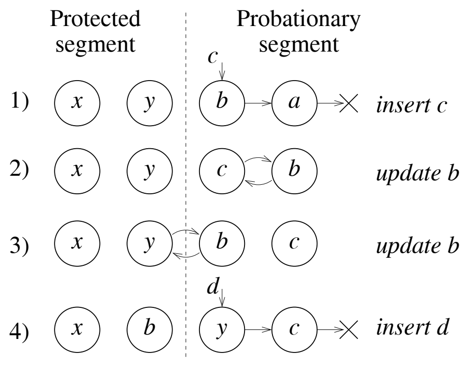

Figure 4: Bucket of NHT managed by S^3-LRU.

5.3 S^3-LRU

The HFC has a limited capacity, insufficient to track every active flow at any time. Therefore, it is important that the HFC stays up to date to contain the state of the most active flows in order to label a major portion of traffic. An early attempt was to include a simple LRU eviction policy to the HFC. However, LRU caches are susceptible to the eviction of wanted entries during a flood of new activity and flows consisting of only a few packets are very common [12]. Single Step Segmented LRU (S^3-LRU) includes a mechanism for dealing with this phenomenon using a variation of the Segmented LRU (SLRU) of [13].

Fig. 4 illustrates that like SLRU, an S^3-LRU policy divides cache into two segments: a probationary segment and a protected segment. When a cache miss occurs, the new flow is added to the front of the probationary segment in the S^3-LRU list and the least recently used flow of this segment is removed (Fig. 4 (1)). Hits (accessed flows) are advanced in the list of a single step toward the front of the protected segment by swapping their position with that of the adjacent flow (Fig. 4 (2)). If the flow is already at the front, it maintains its position. The migration of a flow from the probationary segment to the protected segment forces the migration of the last flow in the protected segment back in the probationary segment (Fig. 4 (3)). The S^3-LRU policy protects the cache against traffic patterns that can flood an LRU cache with flows that will not be reused because these flows will not enter the protected segment. The size of the protected segment is a tunable parameter which depends on the workload. We leave as future work how to select the proper value, and how to adapt it to changes in the workload.

Together, recency and frequency determine the order of flows in the two segments — and ultimately which flow shall be evicted when a new one arrives. Moving each hit by a single position means that it is more likely that only the large flows will compete for the protected segment. This is unlike SLRU, which moves hits to the front of the protected segment and thereby keeps both segments ordered from the most to the least recently accessed flow. Promoting a flow to

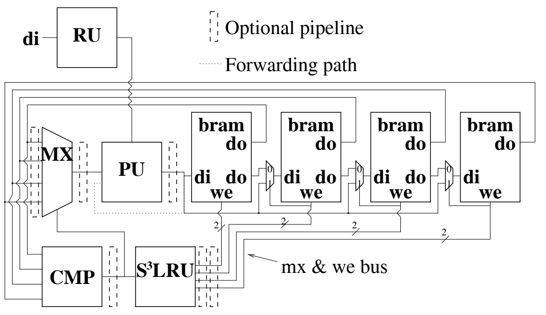

Figure 5: FPGA Implementation of Naïve Hash Table with S^3-LRU replacement policy.

the front of the protected segment as soon as it is accessed weights ordering in favor of recency over frequency, and that penalizes elephant flows.

5.4 Naïve Hash Table with S^3-LRU in FPGA

We consider that an internal on-chip memory is composed of many, equally-sized independently addressable blocks, in case of Xilinx Virtex technology called BlockRAMS. The lay-out of NHT is a row of dual-port BlockRAMs where a bucket is composed of words with the same address, an implementation is displayed on Fig. 5.

Due to the parallel access to all BlockRAMS, all fingerprints in a bucket are available at once and so the lookup is performed in a single clock cycle by comparing each fingerprint with the requested fingerprint (CMP).

Once the position of a corresponding flow state is found it is delivered to the processing unit (PU) via the multiplexor (MX). The PU updates the flow state and stores it back to a new position given by S^3-LRU policy. If the bucket contains more than 4 entries it is convenient to pipeline the design in order to break a potentially critical path caused by a comparator, multiplexor and PU in a chain.

The S^3-LRU can be implemented on-site with no additional memory overhead by moving entries within the bucket (Fig. 4) or can use an additional vector attached to each bucket which keeps the ordering of flow states while the entries stay at the same location.

The on-site implementation, on Fig. 5, needs additional multiplexors in front of memory blocks in comparison to plain NHT. On the other hand, the vector implementation requires additional memory with $n\lceil\log l\rceil$ bits (l – number of entries allocated per one bucket, n – number of all entries in NHT). Despite the memory overhead, vector implementation might prove useful when a flow state is large and is kept apart from the fingerprint in an external memory. In such case, the NHT and the vector memory in FPGA are utilized to lookup a flow state and to maintain the replacement policy as the swapping or shifting data in an external memory is not an option. For example, consider tracking of inter-packet arrival times for the first several packets of each flow. The last packet timestamp and the packet counter can be kept in NHT inside the FPGA while the inter-packet arrival times would be stored in an external memory word by word.

The additional logic consumed by S^3-LRU is negligible. An exemplary output of the S^3-LRU unit for an on-site implementation is given in Table 2. The unit must drive control signals of multiplexors (*mx bus*) and write enable signals of BlockRAMs (*we bus*).

Reqests	mx bus	we bus
insert c	(0,0,1,0)	(0,0,1,1)
update b	(0,0,1,0)	(0,0,1,1)
update b	(0,1,0,0)	(1,1,0,0)
insert d	(0,0,1,0)	(0,0,1,1)

Table 2: An output of an on-site S^3-LRU unit. The setup and sequence of requests corresponds to Fig. 4 with its initial content of a bucket (x,y,b,a).

Either implementations are scalable in size and speed. Both parameters may be improved by allocating additional parallel modules of NHT. Part of a flow fingerprint would determine not only a bucket but also a module to hash in.

The analysis of pipelined NHT reveals two scenarios when additional steps must be taken to maintain stable performance. The three level pipeline would have to be flushed when 2 requests target the same entry in an interval shorter than 5 clock cycles otherwise changes made by the first request would not be accounted for later on.

An exemplar is presented for maintaining flow states of 100 Gbps traffic. A clock cycle of 7 ns allows to update a flow information for every packet even if the traffic is composed of the shortest possible IP packets but at the same time the throughput of any flow in the traffic cannot be higher than 20% of the processing capacity ($\approx$ 20 Gbps). Otherwise the distance between two packets belonging to the same flow would be less than 5 clock cycles (35 ns), and the flow state would not be stored back in the memory before its next retrieval.

This could be overcome by forwarding the PU results back to its input (Fig. 5 (Forwarding path)) and notifying the S^3-LRU unit to update information driving the control signals in the next clock cycle. Subsequently, a Reservation Unit (RU) at the input of NHT can control the distance between requests for the same flow state and may reorder these by inserting other requests in between to make the distance larger than 5 clock cycles or by joining the same requests next to each other so the forwarding path may be utilized.

Another issue arises from requests that target the same bucket but are of two different flows. We solve it by stalling such requests until previous request has not been processed. Such situations are rare since most of the requests are distributed uniformly among all buckets.

5.5 Hardware Setup

The total on-chip memory available in NetFPGA would allow us to accommodate approximately 128 K entries if each entry has a 36 bit fingerprint and a few bits reserved for the application class and the packet counter. Unfortunately, the NetFPGA infrastructure (MAC cores, etc.) consumes over a half of the memory resources. In our prototype, we use the configuration of the HC and the HFC as shown in Table 3. Each bucket in the HC and the HFC contains 8 entries; with a total length of 48 bits for the hash, the false-positive probability of the NHT is negligible for our experiments.

Moreover, our approach works sufficiently well with these limited numbers of table-entries for modest sized organizations. The HC effectively labels a significant number of flows and the HFC gracefully degrades the labeling performance when the number of concurrent flows exceed its capacity. In no case are packets dropped.

Parameters	HC	HFC
Entries	2048	32768
Buckets	256	4096
Fingerprint	40	36
Length of hash	48	48
F.p. probability p_f	7.5×10^{-9}	1.9×10^{-6}

Table 3: Configuration of Host Cache (HC) and Hardware Flow Cache (HFC)

6. PERFORMANCE EVALUATION

The experiments aim at validating our hierarchical packet labeling scheme, and demonstrating the performance advantages of simple and cost-effective hardware.

6.1 Experimental Setup

For this system, the packet and flow rate have more significance than just the data rate. The data path is designed to handle multi-Gigabit/s data streams. However, each packet and flow undergoes a number of operations (e.g., table lookup, feature extraction, classification) that must be able to keep up with packet and flow rates, respectively, so it is important to include realistic traffic in the evaluation methodology. Moreover, as AtoZ derives and uses information of behavior of applications we conclude that real traffic data are fundamental to the system evaluation. To validate our approach, we deployed the system alongside the edge router of a large university campus using a span port to mirror the Internet traffic and tracked the system performance on live data since April 2009.

As an exemplar evaluation in a more controlled environment, we use two data traces for the reported experiments: one from a research institute (R.Inst.), and one from a university campus (Campus) the working dimensions and the methodology for which we obtain ground truth information is described in [14].

Our testbed consists of two high-end server machines which can replay the traffic using `tcpreplay`[5]. We partitioned the traces into two halves, using flow hashing to ensure that packets belonging to the same flow are in the same trace, to allow them being replayed from the two machines with high fidelity. The traffic from the two machines is aggregated in a Cisco C6509 switch to provide high data-rate traffic. The NetFPGA is connected to an optical port on the switch via a media converter. By means of an optical splitter located on the path to the optical port, we also connect a DAG card downstream from the switch so that measurements are not impacted by jitter and delay introduced at aggregation. `tcpreplay` is used to reproduce the timings of the original trace with sufficient fidelity.

The NetFPGA board is hosted on a machine with an Intel Quad Core CPU (2.40 GHz) with 4 GB of RAM, running CentOS Linux 5.0 (kernel version 2.6.18). However, our software implementation is only single threaded.

In order to compare AtoZ with an equivalent software-only solution, we combine our software components with elements from the standard Click distribution that provide packet-receive and transmit functionalities. In this case, we run the software on the same PC hosting the NetFPGA, but we use a high-end dual port NIC (Intel e1000) to receive and retransmit packets, because the NetFPGA is not appropriate as a standard NIC.

[5]http://tcpreplay.synfin.net/

Module	Latency	Mode	Bus width
Eth. Input	608–11952 ns	SF	8 bits
HC	224 ns	CT	64 bits
HFC	208 ns	CT	64 bits
FM	16 ns	CT	64 bits
Out. Pkt Queues	72–1496 ns	SF	64 bits
Eth. Output	16 ns	CT	8 bits

Table 4: Packet processing delay of each module. (SF: store and forward, CT: cut-through)

We point out that our prototype uses research-grade code and many opportunities exist for further improvements to be made. We simply intend to show the performance gap between using a software-only solution and a system that builds on that same software but combines with functionalities implemented in specialized hardware.

6.2 System Capacity

We start by examining whether AtoZ is able to handle a fully loaded link, as our simulation predicted. To test the system's throughput, we generate traffic by replaying the R.Inst. traffic traces at maximum speed. This produces a 1 Gbps data stream. Our experiments confirm that no packets are dropped, even when traffic is sent at maximum speed. We measure the throughput across a 10 s intervals. The mean throughput is about 137 Kpps and achieves a maximum of 173 Kpps, while the software processes on average 1.6 Kpps with a peak of 12 Kpps (duplicated packets).

6.3 System Delay and Packet Loss

We simulate the hardware processing pipeline (Figure 3) to derive the delay introduced by each module. Recall that the Host Cache (HC), Hardware Flow Cache (HFC) and Forward Module (FM) work in cut-through mode as to reduce the overall delay, whereas the Output Packet Queues and Ethernet Input use a store and forward approach. Also note that the delay of each module is cumulative because all packets traverse the same pipeline irrespective of where they are labeled.

The module delays are reported in Table 4. The FM only adds a small delay for rewriting the MAC address based on the label value. The Ethernet Input and Output Packet Queues contribute to most of the delay, although due to a wider bus, the Output Packet Queues delay is significantly smaller compared to that of Ethernet Input. At the expense of hardware complexity, the delays could be reduced by converting these modules to operate in cut-trough mode.

The overall delay is derived using minimum and maximum IP packet lengths of 64 and 1500 bytes respectively. Summing the individual contributions of all modules in the pipeline, we obtain 143 and 1739 clock cycles (125MHz) which corresponds to about 1.1 μs and 14 μs for the minimum and the maximum packet length, respectively.

To measure the actual real-life delay[6], we generate traffic by replaying the captured traces at 300 Mbps, 600 Mbps and maximum speed. The packet-length distribution is of a real traffic mix with most of the packets being longer than 1 KB. Table 5 shows the measured processing delays with standard deviations (which do include buffering and transmission delays). AtoZ maintains a very low delay ($\leq$17 μs) with no

[6]Measurements have been done with DAG cards.

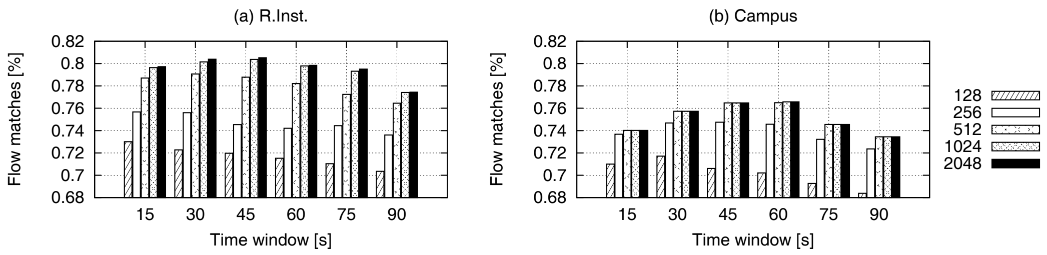

Figure 6: Host Cache efficiency as flow hits in cache.

Data rate (Packet rate)	300 Mbps (51 Kpps)	600 Mbps (107 Kpps)	1 Gbps (137 Kpps)
AtoZ	13 ± 4 μs	16 ± 3 μs	17 ± 2 μs
SW only	304 ± 120 μs	12 ± 25 ms	-

Table 5: Packet processing delay of AtoZ compared to software-only solution.

observed packet loss and host CPU utilization ≤6%. The slightly increased delay of an NetFPGA under a heavy-load is most likely caused by a buffer dynamics of Ethernet Input and Output modules. The software-only system sustains a data rate of 300 Mbps, but its processing delay is already an order of magnitude higher than AtoZ. At 600 Mbps, the software drops about 4% of the packets and the CPU reaches 100% utilization, causing an increase in delay of tens of milliseconds. As packet loss already starts to occur, there is no need to test the software with 1 Gbps. Further tuning might improve the bandwidth of a software solution for the sake of a longer delay due to larger buffers for DMA transfers.

6.4 Packet Labeling

For these experiments, we replayed the traffic traces at their original speed.

Host Cache Efficiency: We evaluate the Host Cache (HC) efficiency by measuring the number of flows that it matches over the total number of flows in each time window. We vary the time window from 15 to 90 s, and we vary the capacity from 128 to 2048 entries by powers of two. Figure 6 shows the average efficiency for some significant time window lengths. The value of U_{th} is 10. We experimented with several values and we found that 10 is a sufficient trade-off between avoiding spurious rules (which happen with lower values) and requiring longer time windows.

As expected, the number of flows matched by the HC increases with a higher HC capacity. The optimum value of the time window is 30 to 45 s for both traces to reach the highest number of matches. Although a longer time window allows establishing with higher confidence which end-points are the most significant and stable over time, extending the time window for too long also reduces the opportunity for applying the rules just derived.

Time-to-bind: An important benefit of the HC is that the system can promptly organize the traffic of new flows involving known hosts: the time-to-bind is zero for all new flows matched by the HC, which means that their packets are labeled from the time the flows start. To demonstrate this improvement, we measure the time-to-bind when the HC is disabled and compare it with the case when it is enabled. Figure 7 shows the time-to-bind histogram obtained for the R.Inst. trace.

Labeling Accuracy: By comparing with pre-determined application ground-truth [14], we evaluate the overall accuracy of the packet labeling process.

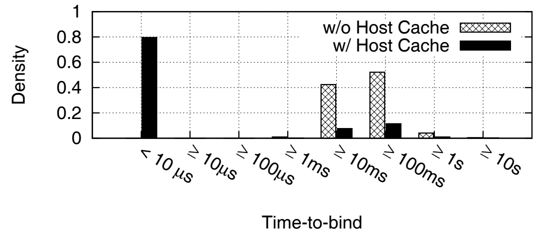

Figure 7: Time-to-bind distribution, with Host Cache vs. without Host Cache.

Two metrics are considered: the flow accuracy and the packet labeling accuracy. To measure the flow accuracy we export the application identification results and HC rules on the host. For the packet labeling accuracy, we use the DAG card to capture only the packets that are not labeled. By combining this with the results above, we determine which packets are correctly labeled.

For R.Inst., AtoZ achieves 99.63% flow accuracy and 99.46% packet accuracy. The accuracy for Campus is also high at 99.59% and 99.36% for flows and packets, respectively.

Load Shedding: Figure 8 presents the temporal evolution of the SFC size for R.Inst. We compare the case of HC disabled with HC enabled. When the HC is disabled, every active flow is present in the SFC, causing greater load on the system (e.g., longer table lookup times) and high utilization of the (limited) HFC resources. In this case, the SFC maintains on average 38.5×10^3 entries which is 17% larger than the HFC. However, when the HC is enabled only about 25% of the active flows need to be maintained in the SFC, reaching a maximum of 25×10^3 entries and less than 77% utilization of HFC.

The two curves in Figure 8 have similar profiles except for point marked (1). This corresponds to the expiration of a certain HC entry due to timeout. As desired, the system periodically expires HC entries even though they are well performing. This is done to maintain accurate traffic classification. Here, the number of flows entering the SFC increases. These are the flows that are matched by the HC until (1), but when the HC entry is removed, the successive 5 packets for these flows are processed by software. AtoZ now has two alternatives: it can label these flows based on the expired HC entry or it can classify the flows using mid-flow features. In our implementation, we chose the simpler solution which is to label the flows using the HC entry. Once the HC entry is removed we still keep a record of the entry in software for one additional flow timeout period. In this case, after one time window the HC management restores the HC entry.

Further, another source of load shedding in our system is represented by the packet filter in HFC which combines with the HC effectively to reduce the traffic processed in software.

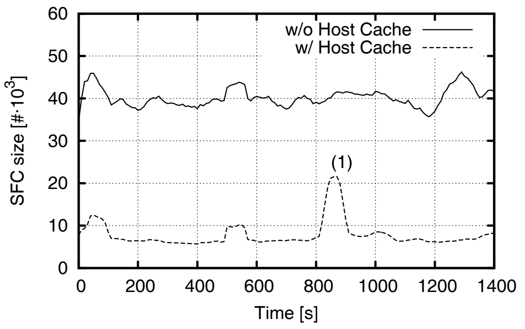

Figure 8: Temporal evolution of SFC size, with HC & without HC.

For our traces, this is reduced to about 0.1% of the original traffic. This result agrees with the findings of other works that also leverage the heavy-tailed nature of traffic. For instance, in [7] the authors reduce the data to less than 6% of the original traffic from their sites by collecting the first 15 KB of each flow. In our experiments, the reduction is even more drastic due to the use of just the first 5 packets and the efficiency of the HC. Disabling the HC increases by 5 times the amount of traffic processed in software.

Tracking most active flows: Using simulations, we evaluate the capability of the HFC to keep track of the most active flows while evicting flows that do not account for significant portions of traffic.

We measure the percentage of large flows tracked by the HFC using different replacement algorithms – Least Recently Used (LRU), Segmented LRU (SLRU) and S^3-LRU.

We look separately at how well the algorithms perform for three reference groups: very large flows (above one thousandth of the link capacity), large flows (between one thousandth and a tenth of a thousandth) and medium flows (between a tenth of a thousandth and a hundredth of a thousandth). We experiment with just 1024 entries in the cache to guarantee that there are more active flows than those that can be tracked at any time. We use a measurement interval of 5 s and probationary segment size set to 70% of the entries in each bucket as it was determined to perform the best after a full space search.

Table 6 presents the results averaged over all runs and measurement intervals. Our replacement algorithm is able to obtain lower values of unidentified large flows for all the tested conditions (not reported due to space limit).

We note that many recent approaches for tracking large flows (e.g., [10]) are based on a compact filtering data structures that allow to identify and account large flows only after they pass the filter threshold. However, our application specifically requires the possibility of assigning a state to all flows, even before they could pass the filter.

7. DISCUSSION

Robustness

The robustness of the whole system is guaranteed by the hardware fast data path, in which each component is designed to support full line-rate packet switching.

System failures may impact other traffic organizer implementations due to resource exhaustion or system-overloading. In our system, there is no interruption to the fast data path which serves as the default path for all data, so that even

Algorithm	Group (flow size)		
	$> 0.1\%$	$0.1\ldots0.01\%$	$0.01\ldots0.001\%$
S^3-LRU	0.14	0.24	0.88
SLRU	17.10	10.20	35.50
LRU	23.53	12.60	41.72

Table 6: Unidentified large flows [%] for S^3-LRU vs. SLRU and plain LRU using 32 buckets each of 32 entries.

in exceptional circumstance only the software system can be overloaded. In the worst case only a certain portion of packets are labeled. We could conjecture that the resulting performance will still be an improvement over no device, since at least some traffic labeling would take place. While the current approach has been effective in deployment, a full characterization of the cache hierarchy is left for future work. In common with many high-end network device designs, e.g., NIDS[15], the hierarchical packet labeling scheme also provides more resilience against attacks which lead to poisoning of the flow cache.

Quality of Experience (QoE)

QoE provides a motivation and our prototype is useable as such a device. We currently perform aggregate rate-limiting in deployment and while space limitations prevent discussing this in depth, AtoZ flexiblity is suited to a wide range of approaches such as application-specific rate-limiting, and other protocol-specific methods. Further, AtoZ can provide a flexible classification of network-traffic for other systems such as diffserv and MPLS.

Extensibility

The traffic organizer presented uses only rules automatically learned from the flow classifier. However, thanks to its modular structure, it is readily extensible to allow input information from other sources such as an organization's NIDS or input from a manual interface, or to provide network utilization information to specific traffic-handling systems such as firewalls and NIDS implementations.

Scalability

The size of on-chip memory is the dominant factor for the system scalability. The current constraints of on-chip memory have meant that the system worked well for smaller institutions ($\approx$1,000 hosts) but was not best-suited to the largest enterprise (15,000 users and close to 100,000 hosts).

The NetFPGA board is equipped with 4 MB of SSRAM memory that is currently dedicated to the packet buffers and queues. We estimate half of the memory could accommodate almost 500 K entries for the HFC.

However, scaling the system to higher speeds (i.e., $\geq$ 10 Gbps) requires us to keep the caches on the FPGA in order to support link-rate processing. At the same time, the frequency and the bus width of the whole processing pipeline must be designed to cope with 20 Gbps of network traffic. The most likely option seems to be the 128 bits wide bus synchronized to 156 MHz design frequency. To achieve such configuration, we may extrapolate from Virtex-II-Pro 50 to a Virtex-5-LX155T and also to a larger Virtex-5-SX240T respectively providing near twice and four times the on-chip memory capacity. Extrapolating results from our previous experiments, such resources provide the necessary groundwork to process network traffic at and above 10 Gbps.

Implementation

Our current implementation is based on software extensions to Click and makes extensive use of the NetFPGA.

The NetFPGA was available to us and so its use as our implementation/evaluation vehicle was ideal. It is a flexible platform and real multi-Gigabit/s solution which permits rapid prototyping of our system and demonstrates the feasibility of the hybrid architecture to fulfill the design objectives. Further, the implementation we have made is publicly available and may be ported to existing systems (e.g., Netronome and NetCOPE), to provide 10 Gbps implementations at reasonable cost.

For future implementations, one tempting approach is to extend the Linux netfilter to use the NetFGPA by adding hardware implementations of appropriate functional modules, as we have done in the AtoZ traffic organizer.

8. RELATED WORK

Our work on the AtoZ automatic traffic organizer relates to a wide set of fields ranging from systems-architectures to application-identification based upon machine-learning.

We are indebted to the creators of the Click modular route [8], and extend this idea with an approach that preserves functionality between modules. While this does not permit the same level of *plug'n'play* that Click provides, our approach still permits high-performance hardware modules to be co-designed alongside software implementations.

The architecture of AtoZ shares a few common characteristics with other FPGA-based architectures that "shed load" for network-intrusion detection and prevention systems (e.g., SnortOffloader [16] and Shunt [17]), in particular, the way that it distributes the workload and functionalities between hardware and software. These approaches are specifically designed to offload the *static* subset of traffic that is large in volume but of little interest to intrusion detection and prevention systems. In contrast, AtoZ aims at *automatically* organizing the entire traffic on the link, and tackles different challenges associated with traffic organizing.

9. CONCLUSION

We have described the design and implementation of the AtoZ traffic organizer: a small, intelligent and high-speed network device. It enables seamless, application-specific traffic management on edge-network, benefiting from a highly-efficient packet-labeling mechanism based on intelligent behavioral flow identification, and high performance packet processing using NetFPGA. We deal with several technical challenges: how to support the entire system functionality under maximum throughput, without delay or packet loss; while incorporating effective intelligent traffic classification.

Furthermore, we showed the feasibility of a simple, cost-efficient hardware-software hybrid implementation that facilitates seamless and sophisticated traffic management operations on multiple Gigabit-rate links.

We are making our implementation available to the community and at the NetFPGA project page.

Limitations and Future Work

Our implementation is an early-day prototype, and while deployment experience has been positive, we can identify several areas for future work.

One avenue to be explored is that our implementation could logically be integrated into an existing access-router; and while current practical limitations make this a challenge for a NetFPGA-based solution, current FPGA technology in combination with suitable, modest, quantities of SSRAM can easily accommodate such a set of logic and caches.

We also note that our work does not provide full traffic-accounting functionality at line rate. However, there is no reason (aside from NetFPGA gate-resource limitations) that AtoZ could not be integrated with the NetFlow probe also developed on the NetFPGA.

Acknowledgements

This work is partially supported by the research intent MSM-383917201. This work is supported by the EPSRC through grant GR/T10510/02. We also thank the numerous network-operations groups who have supported this research, they include JaNet, Dante, KAREN, AT&T and GARR. We thank the anonymous reviewers for their many helpful comments and suggestions, we also thank Jon Crowcroft, David Miller, Steven Hand, Damien Fay, and Stephen Kell for their helpful comments on drafts of this work.

10. REFERENCES

[1] M. Roesch. Snort — Lightweight Intrusion Detection for Networks. In *Proceedings of USENIX LISA'99*, 1999.

[2] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[3] J.W. Lockwood *et al.* "NetFPGA–an open platform for gigabit-rate network switching and routing". In *IEEE International Conference on Microelectronic Systems Education (MSE'07)*, 2007.

[4] L. Bernaille *et al.* Early application identification. In *Proceedings of the ACM CoNEXT'06*, December 2006.

[5] W. Li *et al.* Efficient application identification and the temporal and spatial stability of classification schema. *Computer Networks*, 53(6):790–809, Apr 2009.

[6] M. Dusi *et al.* Tunnel Hunter: Detecting application-layer tunnels with statistical fingerprinting. *Computer Networks*, 53(1):81–97, Jan 2009.

[7] G. Maier *et al.* Enriching network security analysis with time travel. In *Proceedings of ACM SIGCOMM'08*, 2008.

[8] R. Morris *et al.* The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[9] A. Broder & M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–609, 2003.

[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[11] A. Kumar *et al.* Space-code bloom filter for efficient per-flow traffic measurement. In *Proceedings of IEEE INFOCOM*, Mar 2004.

[12] R. Pang *et al.* Characteristics of internet background radiation. In *Proceedings of IMC'04*, 2004.

[13] R. Karedla *et al.* Caching strategies to improve disk system performance. *Computer*, 27(3), 1994.

[14] M. Canini *et al.* GTVS: Boosting the collection of application traffic ground truth. In *Proceedings of TMA'09*, May 2009.

[15] M. Attig and J.W. Lockwood. SIFT: Snort intrusion filter for TCP. In *Proceedings of the 13th Symposium on High Performance Interconnects (HOTI'05)*, 2005.

[16] H. Song *et al.* Snort offloader: A reconfigurable hardware NIDS filter. In *Proceedings of FPL'05*, 2005.

[17] J. Gonzalez *et al.* Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention. In *Proceedings of CCS'07*, 2007.