# TRACKING ELEPHANT FLOWS IN INTERNET BACKBONE TRAFFIC WITH AN FPGA-BASED CACHE

*Martin Zadnik*[§], *Marco Canini*[†], *Andrew W. Moore*[‡], *David J. Miller*[‡], *Wei Li*[‡]

[§] FIT, Brno Univ. of Technology, CZ
[†] DIST, University of Genoa, Italy
[‡] Computer Laboratory, University of Cambridge, UK

## ABSTRACT

This paper presents an FPGA-friendly approach to tracking elephant flows in network traffic. Our approach, Single Step Segmented Least Recently Used ($S^3$-LRU) policy, is a network traffic-friendly replacement policy for maintaining flow states in a Naïve Hash Table (NHT). We demonstrate that our $S^3$-LRU approach preserves elephant flows: conservatively promoting potential elephants and evicting low-rate flows in LRU manner. Our approach keeps flow-state of any elephant since *start-of-day* and provides a significant improvement over filtering approaches proposed in previous work. Our FPGA-based implementation of the $S^3$-LRU in combination with an NHT suites well the parallel access to block memories while capitalising on the retuning of parameters through dynamic-reprogramming.

## 1. INTRODUCTION

Conventionally, flow analysis tools such as Cisco NetFlow[1] form an important component in the administration of an operator's network. Amongst other things, they are used for traffic engineering, making forecasts of future provisioning requirements, and dealing with network anomalies.

High speed links carry far more traffic than can be handled by the computing resources available within a router, so tools like NetFlow restrict themselves to simple statistics of active flows based on a sample (typically 1:1000) of all traffic. As volumes grow, either resources must grow to match, or sampling rates must decrease. Since volumes are growing faster than, for example, memory speed and density, sampling rates must decrease, which in turn reduces the quality — and therefore usefulness — of data collected [1].

The ability to identify which applications generate the traffic carried is immensely desirable, but that ability is limited by the information lost due to statistical sampling methods. Sampling can even distort data to the point that incorrect conclusions are reached [2].

[1]`http://www.cisco.com/go/netflow`

The parallelism inherent within an FPGA [3], along with low delay and wire speed processing [4] makes the FPGA a promising platform in which to implement a more robust flows analysis solution.

In common with the observation that a small number of destinations can account for a large amount of network traffic [5], measurement studies, e.g. [6] have showed that a small percentage of "heavy-hitters" account for a large share of the traffic and this has become critical in the design and engineering of current and future networks [7]. By using the algorithms introduced by Estan and Varghese [8] for identifying elephant flows, only packets which belong to these flows are recorded in a flow state memory. Estan [8] also define a term elephant flow which we adopt in our work.

Typically, a hash-based filter is used to select for elephant flows. Each flow owns a list of counters in the hash table, and each counter may be shared between many flows. With such a scalable approach, it becomes feasible to perform flow measurements, even for backbone links. Moreover, with the filter being so selective, the flow table and the filter are small enough to reside within internal FPGA RAM resources.

The disadvantage of this approach is that a flow will only be accounted for once its volume has passed the threshold, and no state can be assigned to the flow until that time.

We present a novel approach to elephant flows detection which allows for preserving the entire flow state from the first packet — thus making the early packets available for specific flow analysis, such as [9].

We achieve this by means of a flow record cache and a novel cache replacement scheme. Conventional LRU (Least Recently Used) makes the assumption that the probability of an item access decreases with time since its last access. This has the unfortunate consequence that a burst of new items may cause the eviction of frequently used items.

The rest of this paper shows how $S^3$-LRU addresses this shortcoming by combining historic access frequency with conventional LRU and the distinctive patterns of Internet traffic workloads, and how $S^3$-LRU is apt to be implemented within FPGA technology.

## 2. BUILDING AN ELEPHANT FLOW CACHE

The goal is to design a flow cache that is able to maintain flow state for elephant flows since their very beginning. Our design takes advantage of FPGA technology to keep the pace with the increasing speed of backbone links.

### 2.1. Addressing scheme

While there exists many types of high-performance addressing schemes (*e.g.* Fast Hash Table [10], Cuckoo Hashing [11], and Tree Bitmap [12]), we adopt the basic Naïve Hash Table (NHT) [13] because NHT supports parallelisable fast lookups in multiple memories without additional memory overhead, and new replacement policies can be added with minimal effort.

The concept of the Naïve Hash Table is based on using a hash to divide a search space into disjunctive buckets with approximately the same number of entries in each bucket. Searches are made by computing a hash over a request descriptor to locate a bucket, which is then searched sequentially until the target entry is identified.

Since internal FPGA memory is limited, we trade the false-positive rate for a greater number of table entries. Instead of the full flow descriptor, each entry contains only a small fingerprint of the original IP 5-tuple. The increased probability of two different flows colliding with the same hash value can be mitigated by properly dimensioning the bit-length $b$ of the fingerprint.

The probability of false positives $p_f$ can be estimated by the solution to the birthday problem [14]. For $2^h$ buckets, the bit-length of the hash 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 total number of distinct hashes and $n$ is the number of entries in use. Table 1 gives the false positive rate for several configurations of 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 ($p_f$)

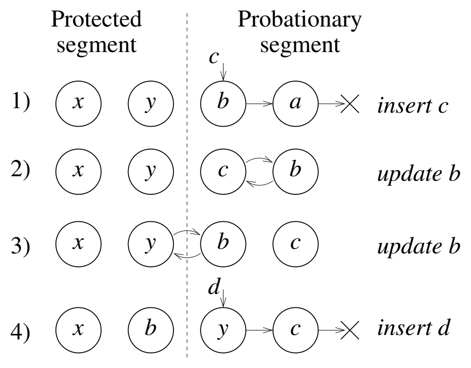

**Fig. 1**. Bucket of NHT maintained by S$^3$-LRU policy.

### 2.2. S$^3$-LRU

LRU caches are susceptible to the eviction of wanted entries during a flood of new activity. Flows consisting of only a few packets are very common, colloquially known as *Internet background radiation* [15]. S$^3$-LRU includes a mechanism for dealing with this phenomenon using a variation of Segmented LRU (SLRU). SLRU was introduced in [16] and it is a variation on LRU.

Similarly to SLRU, an S$^3$-LRU policy divides cache into two segments: a probationary segment and a protected segment, as shown in Fig. 1. 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. 1 (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. 1 (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. 1 (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 the front of the protected segment as soon as it is accessed weights ordering in favour of recency over frequency, and that penalizes elephant flows.

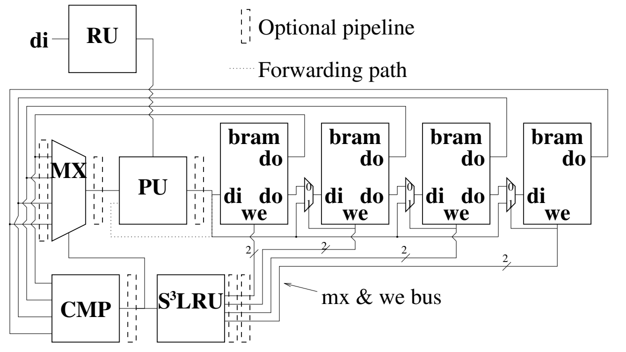

**Fig. 2**. Implementation of Naïve Hash Table with S$^3$-LRU replacement policy in FPGA.

## 2.3. 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. 2.

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 (as depicted on Fig. 1) or with 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. 2, 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. 1 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 examplatory case 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 link capacity (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. 2 (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.

## 3. EVALUATION

We prototype the system on NetFPGA platform [4]. Therefore, the elephant flow cache (the on-site solution) was written in Verilog and synthesized (XST 10.1) along side with other components for Virtex-II-Pro-50 (speedgrade -7). The NetFPGA on-chip infrastructure (MAC cores, DMA engine, etc.) already consumes over a half of the memory and logic resources. In order to save on-chip resources, the extensions to overcome conflicts in the cache were omitted without any consequence since the incoming packet rate is anyway limited by the bandwidth of available network interfaces.

| Configuration | # BRAMs | # Slices | Frequency |
|---|---|---|---|
| 512 × 32 × 32 | 33 | 2394 | 154 MHz |
| 512 × 32 × 40 | 65 | 2560 | 143 MHz |
| 1024 × 16 × 32 | 33 | 1117 | 154 MHz |
| 1024 × 16 × 40 | 65 | 1708 | 154 MHz |

**Table 3**. FPGA resource consumption by the elephant flow cache. Configuration: # of buckets × # of entries per bucket × fingerprint bit-length.

The achiavable frequency and the resource consumption of several configurations were observed as summarized in Table 3. These results were acquired using a separate round of the synthesis process when the cache was synthesized without other components that might influence the reported frequency.

Clearly, the parameters reported by the synthesis process are only estimates since physical placement on the chip is not yet known as the interactions with other components have not been accounted for.

Therefore, we place the flow cache in the design that already accounts for 131 (56%) BRAMs and 16789 (71%) Slices. The design itself can placed and routed while meeting the constraint given by 8 ns clock cycle (125 MHz). We gradually increase the size of each bucket and the bit-length of a fingerprint while maintaining a constant number of total entries. We are interested in a situation when a critical path is caused by the elephant flow cache rather than by any other component (Table 4). Note please that the entries contain only a fingerprint so the limiting points might be revealed. If the additional data are added into an entry the overall number of entries decreases proportionally which is the case presented during our experiments with S$^3$-LRU .

The results show that the size of a bucket is limited to a maximum of 32 entries with 32-bit fingerprint. The critical path is then caused by the comparator of a requested fingerprint with multiple fingerprints from a bucket. For a high end system that processes 10 Gbps links and above, we expect faster and larger FPGA such as Virtex 5 with more available memory capacity and better timing performance.

| Bucket size | Bit-length of fingerprint $b$ | | |
|---|---|---|---|
| [entries] | 24 | 32 | 40 |
| 16 | 125 (3) | 125 (5) | 125 (11) |
| 32 | 125 (39) | 125 (96) | 116 |

**Table 4**. Achieved frequency of elephant flow cache with 16K entries in fully routed design. Value in brackets gives the time (in minutes) that the router took to route the design.

We use two 30 min traces of Internet traffic: one (HALL) from the 1 Gbps edge link of a large university campus and an anonymized, unidirectional trace (CENIC) from NLANR (collected at the 10 Gbps CENIC HPR backbone). Table 5 summarizes the working dimensions of our traces.

| Trace | Duration | Packets | Bytes | Avg. Flows |
|---|---|---|---|---|
| HALL | 30 min | 57 M | 35 G | 14.9 K |
| CENIC | 30 min | 58 M | 54 G | 4.4 K |

**Table 5**. Working dimensions of our traces.

In our experiments we define flows in the same manner as NetFlow: by the 5-tuple of source and destination IP address and port and the protocol number. As in [8], we chose to use a measurement interval of 5 s in all our experiments. We assume our implementation can use about half Mbit of memory (1024 entries[2]).

Our goal is to measure the percentage of large flows tracked by the flow cache with different replacement algorithms. We adopt the definition of large flows from [8]. 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).

| Algorithm | Group (flow size) | | |
|---|---|---|---|
| | > 0.1% | 0.1. . . 0.01% | 0.01. . . 0.001% |
| 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 flows [%] for S$^3$-LRU vs. SLRU and plain LRU using 32 buckets each of 32 entries.

Table 6 presents the results averaged over all runs and measurement intervals. By several experiments we have estimated the proper size of protected and probationary segment being 30% and 70% respectively. Our replacement algorithm is able to obtain lower values of unidentified flows for all the tested conditions.

[2]As in NetFlow, we use 64 bytes per entry.

## 4. RELATED WORK

Our work is inspired by the traffic accounting scheme of Estan and Varghese [8]. They proposed a novel byte-counting algorithm based upon a Multistage filter, a structure derived from counting Bloom filter, which focuses upon the identification and monitoring of a small number of elephants. However, the filter method is not suitable for assigning a state to a flow until it has reached the threshold.

The efficient storage of a flow information has been a research interest for many years. Many indexing schemes were suggested (e.g., [12]) but the most popular ones are variations of hash tables [10, 11].

The LRU replacement is widely used for management of virtual memory, file buffer caches, and data buffers in database systems. Many efforts have been made to address its inability to cope with access patterns with weak locality. For example, Jiang and Zhang [17] have proposed a novel replacement algorithm called Low Inter-reference Recency Set (LIRS) which uses the number of other distinct blocks accessed between two consecutive references to a block to predict the next reference time. However, to the best of our knowledge, this is the first work to propose a variant of LRU that specifically considers the problem of elephant flows identification for network flows.

## 5. CONCLUSION

Most of the network traffic processing requires maintaining a state information on a per-flow basis which leads to a usage of external memories to store all the concurrent flows. Inevitably, the time to access data in external memories is longer than the time to access the on-chip block memory, which poses a bottleneck in current usage of FPGA for high speed monitoring.

We approach our solutions by presuming that for many network applications it is sufficient to maintain flow-state for the , large flows only. This paper presented a Single Step Segmented Least Recently Used ($S^3$-LRU) policy, which is a network traffic-friendly replacement policy for maintaining flow states in a NHT.

Our concept of the $S^3$-LRU preserves information of elephant flows while conservatively promoting potential elephants and evicting low-rate flows in an LRU manner. Using $S^3$-LRU for identification of elephants allows our approach to maintain the flow state since the system-start and is our key difference to the filtering approach proposed in previous work. We offered two possible implementations of $S^3$-LRU in FPGA and showed that $S^3$-LRU in combination with NHT mapped very well onto FPGA architecture where it benefits from parallel access to block memories and the possibility of tuning parameters through dynamic-reprogramming the chip.

## 6. REFERENCES

[1] N. G. Duffield, "Sampling for passive internet measurement: A review," *Statistical Science*, vol. 19, no. 3, pp. 472–498, 2004.

[2] H. Jiang *et al.* , "Lightweight application classification for network management," in *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management (INM'07)*, 2007, pp. 299–304.

[3] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on fpgas," in *FPGA '04*, 2004, pp. 223–232.

[4] J. Naous *et al.* , "Netfpga: reusable router architecture for experimental research," in *PRESTO '08 co-located with Sigcomm 2008*, 2008, pp. 1–7.

[5] G. Cormode *et al.*, "What's hot and what's not: tracking most frequent items dynamically," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 249–278, 2005.

[6] A. Feldmann *et al.*, "Deriving traffic demands for operational ip networks: methodology and experience," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 265–280, 2001.

[7] K. Papagiannaki *et al.* , "Impact of flow dynamics on traffic engineering design principles," in *Proceedings of INFOCOM 2004*, Hong Kong, Mar. 2004, pp. 2295–2306.

[8] 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, no. 3, pp. 270–313, 2003.

[9] W. Li *et al.* , "Efficient application identification and the temporal and spatial stability of classification schema," *Computer Networks*, vol. 53, no. 6, pp. 790–809, Apr 2009.

[10] H. Song *et al.* , "Fast hash table lookup using extended bloom filter: an aid to network processing," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 181–192, 2005.

[11] L. Devroye and P. Morin, "Cuckoo hashing: further analysis," *Inf. Process. Lett.*, vol. 86, no. 4, pp. 215–219, 2003.

[12] W. Eatherton *et al.* , "Tree bitmap: hardware/software ip lookups with incremental updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[13] T.H. Cormen *et al.* , *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[14] D. Wagner, "A generalized birthday problem," in *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer-Verlag, 2002, pp. 288–303.

[15] R. Pang *et al.* , "Characteristics of internet background radiation," in *ACM Internet Measurement Conference*, Oct 2004, pp. 27–40.

[16] R. Karedla *et al.* , "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[17] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 939–952, 2005.