Architectures for Packet Classification Caching

Kang Li
College of Computing
Georgia Institute of Technology
Atlanta, Georga, USA
kangli@cc.gatech.edu

Francis Chang Damien Berger Wu-chang Feng
Departmet of Computer Science and Engineering
Oregon Graduate Institute at OHSU
Portland, Oregon, USA
{francis,damien,wuchang}@cse.ogi.edu

Abstract— **Emerging network applications require packet classification at line speed on multiple header fields. Fast packet classification requires a careful attention to memory resources due to the size and speed limitations in SRAM and DRAM memory used to implement the function. In this paper, we investigate a range of memory architectures that can be used to implement a wide range of packet classification caches. In particular, we examine their performance under real network traces in order to identify features that have the greatest impact. Through experiments, we show that a cache's associativity, replacement policy, and hash function all contribute in varying magnitudes to the cache's overall performance. Specifically, we show that small levels of associativity can result in enormous performance gains, that replacement policies can give modest performance improvements for under-provisioned caches, and that faster, less complex hashes can improve overall cache performance.**

Index Terms **— Caching, Hash, Network Processor, Packet Classification**

I. Introduction

There are innumerable network devices and applications that require fast packet classification. Examples include edge routers performing priority marking [1], web switches, network address translators [2], firewalls, accounting mechanisms within routers. The packet classification process determines which flow [3] a packet belongs to based on one or more fields in the packet. Typical classifications being done today involve fields of a packet's header including the source IP address, the source port, destination IP address, destination port, and protocol. With the increasing speeds of modern networks, the decreasing packet sizes of emerging network applications [4] and the economic realities facing equipment vendors, it is of paramount importance to deliver fast packet classification functionality at a very low cost.

Over the last several years, there has been a large amount of work done on packet classification [5] [6] [7] [8]. The problem itself is a generalization of the one-dimensional IP route lookup problem, but is much harder and requires considerably more resources to perform. It has been well-established [9] that memory access delays limit the classification speeds. While the lookup algorithm itself can be implemented in hardware, the dynamic nature of the classifying rules requires that the classification table be stored in memory. As memory speeds have not kept pace with the rest of the hardware advances, classification speeds are limited by memory access latency. State-of-the-art memory access latencies are about 50 to 60 nsec in DRAM, 5 to 20 nsec in SRAM, and 1 to 2 nsec in on-chip SRAM [9]. Even with on-chip SRAM to store the classification table, the classification process can only afford about 4 memory lookups at 40Gbps. Unfortunately, the best solutions to this problem still require a significantly higher number of memory accesses.

The best way to speed this classification lookup is to avoid doing it by caching previous classification decisions and using them directly. Caching improves lookup speeds by taking advantage of the locality in the traffic [3]. While full classification algorithms require multiple memory accesses, cache lookups can be performed using a single memory access. Unfortunately, while IP route caches only need to scale to the number of routes, packet classification caches must scale up to the total number of flows. Because of this, packet classification caches must be reasonably sized in order to maintain high hit rates. One solution to this problem is to use a large amount of hardware resources. However, with the emerging network processor platforms that are focused on high-performance at a low-cost [10], it is becoming imperative to develop solutions that maximize performance while minimizing the amount of silicon resources consumed. In addition, resources that are not used to implement such a cache can be utilized to speed up the full packet classification algorithm that must be run on a cache miss.

In this paper, we attempt to answer the following question: *Given a limited silicon resources, how should we*

This material is based upon work supported by the National Science Foundation under Grant EIA-0130344, ITR-0121643, and the generous donations of Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Intel.

choose the aspects of a cache architecture, such as associativities, replacement algorithms, and hash functions, to implement fast packet classifications?

The main consumption of silicon resource by a classification cache is the cache memory and the logic to index, store and verify the cache results. More specifically, to determine how to best use the limited resource, we investigate the performance of different cache associtivities, cache replacement policies and hash functions under several real network traces, in order to identify features that have the greatest impact on performance. By performing multi-factor experiments using a cache simulator on real traces, we examine the impact that these architecture aspects have on the performance of packet classification caches over modern network workloads.

This paper is structured as follows. Section II describes the related work on packet classifications with and without caching. Section III presents our approach to evaluating ways of improving cache performance while keeping a small cache size. Section IV reviews the general structure of a packet classification cache by comparison with a traditional CPU memory cache, and section V evaluates different aspects of cache architectures. Section VI concludes the paper with a recommendation.

II. Related Work

There have been a number of related studies in this area. Strong temporal localities have been observed in the Internet back to the early NSF backbone. A study of the NSFNET backbone by Claffy [3] in 1994 showed that caching has significant potential to improve route lookup performance. Recent studies [11] [12] [13] show that the arrival of a packet on an Internet link implies a very high probability of the arrival of another packet with the same flow identifier.

Various packet classification algorithms have been proposed in the last few years, and Gupta et.al. [14] present a survey for various classification algorithms. For implementations, Lakshman et al. [6] use a hardware implemented packet classification to achieve 1 million lookups per second for a few thousand filter rules. Srinivasan et al. [15] propose an algorithm that can achieve 2.47 million lookups per second for less than 50 filtering rules. All of these numbers are measured by assuming static filtering rules so the classification can be fully implemented in hardware.

Routing tables are dynamically updated and have to be saved in memory, and thus the lookup can not be purely implemented in hardware. Patridge et al. [16] place an Alpha processor on each line card and use its $96KB$ L2 cache to fit the entire route cache, showing that a 5000 entry cache can achieve a higher than 90% hit rate. Studies for packet classifications based on flow identifier show similar improvement as route lookup by using a cache. Using a custom ASIC implementation, Xu et. al. [5] achieve a hit rate over 90% for a layer-4 classification.

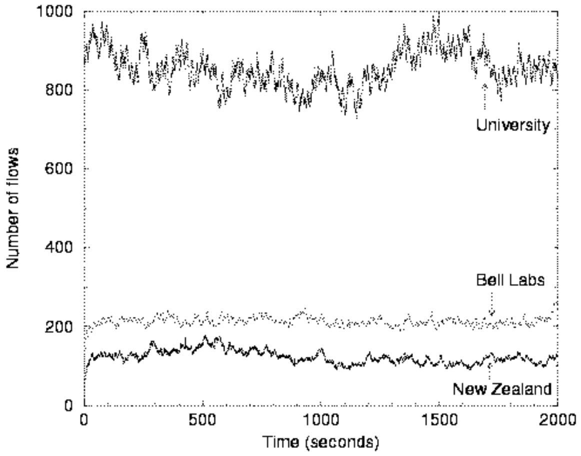


Fig. 1. Flow volume in traces

Our study differs from the above in that we address the following:

- A range of cache architecture aspects
- Validation using recent traces

III. Approach

The method we use to evaluate caching performance is to use trace-driven simulations. In particular, we use real traffic traces as inputs to a highly parameterizable cache simulator that can be configured to vary the cache's size, associativity, replacement policy, and hash function.

A. Trace Data Sets

We use three sets of data for our evaluation. Among these traces, the BELL and Auckland data sets are obtained from the NLANR PMA project [17] and the OGI data set is obtained from an OC-3 link that connects a university campus network to the Internet. two In these statistics, a flow is considered to be a uni-directional entity, defined by a unique 5-tuple (source IP address, destination IP address, source port, destination port, protocol). We call the 5-tuple a FlowID. A flow begins when the first packet bearing a unique FlowID arrives. A flow ends when the last packet is observed, or after a 60 second timeout. The number of active flows in each trace are shown in Figure 1.

B. Packet Classification Cache Simulator

To conduct our packet classification caching study, we implemented a trace-driven cache simulator (PCCS) that allowed us to test a number of cache architectures and sizes. PCCS simulates a packet classification cache in software. It

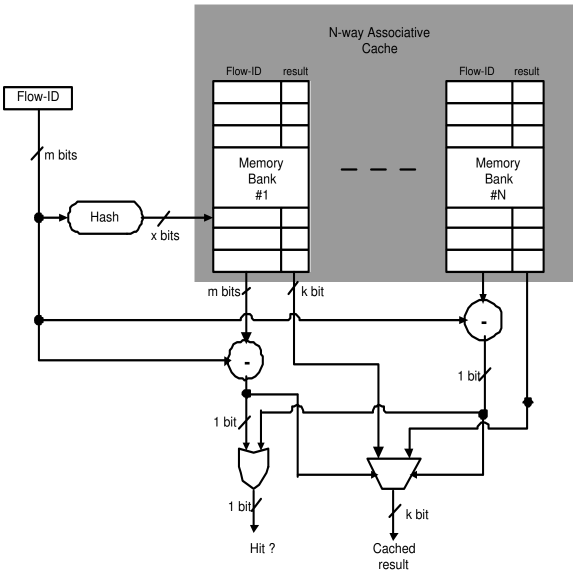


Fig. 2. Cache Architecture

takes traffic traces (in tcpdump or NLANR data format) as input, and calculates hit rates based on the cache configuration. PCCS supports a large set of cache parameters that the user can set such as cache size, associativity, replacement policy, and hash function. In addition, PCCS can also report statistical information about the flows in the input trace. The PCCS simulator is an open-source tool that is available for download [18].

IV. Cache Architecture

A packet classification cache is intended to give classification results in a speed close to the fastest memory speed, so that the classifications can be conducted at link speed. It achieves its low-latency lookups by remembering past classification results in high-speed memory, and first searching these results upon the arrival of new packets.

Figure 2 depicts a general packet classification cache, which is very similar to a CPU memory cache [19], in that both take advantage of locality. In this packet classification cache, the cache memory is an N-way set associative cache, which splits the cache memory into N memory banks. Each memory bank is a directly mapped cache that is addressable by the output of the hash function. For an N-way set associative cache, every input FlowID selects N memory entries, one from each memory bank. Each cache entry contains an $m - bit$ FlowID and a $k - bit$ classification result[1]. One difference from a CPU memory cache and a packet classification cache is that a packet classification cache does not need a valid bit associated with each unit, because an invalid FlowID, such as 0, can be used for this purpose.

Once one cache unit has been selected from each of the N memory banks, comparisons are made to find out if any of N cached FlowIDs match to the input FlowID. If a match is found, then there is a cache hit, and the corresponding classification result is selected. If there are no matching FlowIDs, then a cache miss happens. A full classification must start to search for the input FlowID. The classification result is then forwarded back to the packet classification cache, and a cache replacement may happen depending on the replacement policy.

Various cache aspects of this cache architecture can affect its performance. In the next section, we evaluate the impacts of three important aspects: cache associativities, cache replacement algorithms, and hash functions.

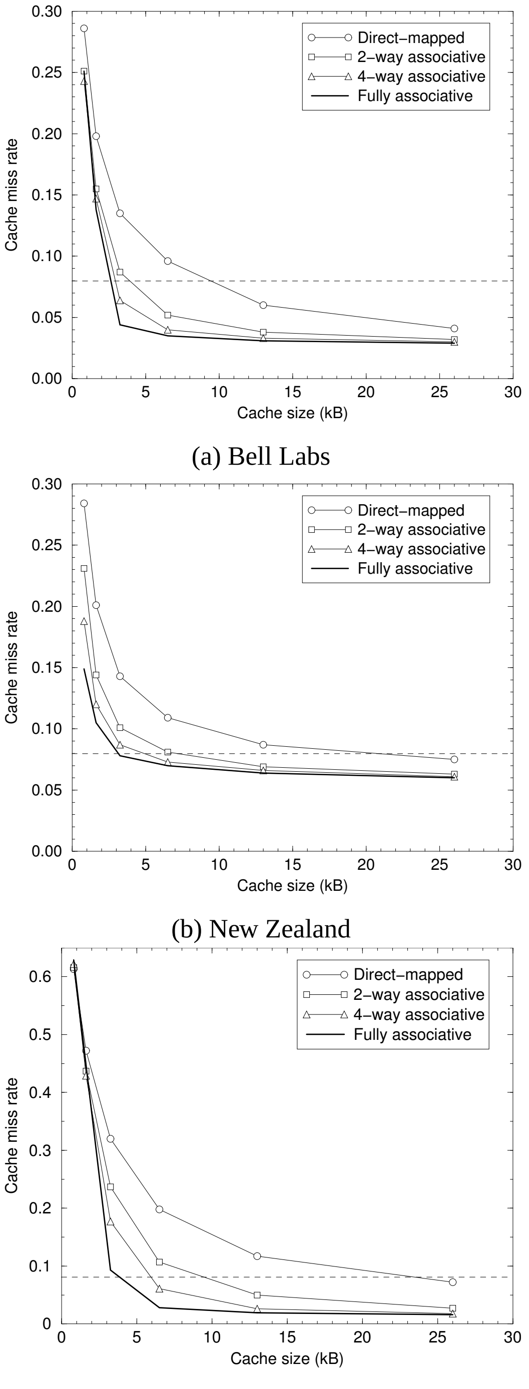
V. Evaluation

A. Cache associativity

Increasing associativity is a widely used technique in CPU memory caches to reduce the impact of conflict misses and increase hit rate. In this section, we first study the impact of increasing associativities on packet classification caches, and estimate the best level of associativities that a packet classification cache should have. The study is performed by running cache simulations with various traffic traces, and observing the impact of various associativities on cache performance.

Figure 3 plots the cache miss rates over a range of cache sizes and associativities on our three workloads. The cache uses a Least-Recently-Used (LRU) replacement policy and a SHA-1 hash function. The cache miss rate is plotted since it directly determines the performance required from the packet classification algorithm being used for a given target packet rate. Doubling the miss rate effectively doubles the required performance of the full packet classification lookup. The figures plot the storage costs of the cache assuming a standard, 13-byte IPv4 FlowID is stored per entry. Note that for supporting IPv6-based packet classification caching, the storage costs will be significantly larger. The graphs show the performance of a direct-mapped cache, a 2-way associative cache, a 4-way associative cache, and a fully associative cache using a LRU replacement algorithm. Figure 3 shows small increases in levels of associativity greatly improve the performance of the cache with a 4-way associative cache, closely matching the performance of a fully associative cache. The saving in storage costs between a direct-mapped and a 4-way associative cache are quite significant. In particular, depending on the trace used, a direct-mapped cache requires between two to three times more storage than a 4-way associative cache in order to maintain a miss rate

[1] The classification result is at least 1 bit for a packet filter, but could be multiple bits for example, to record the service levels of diff-serv applications.



(a) Bell Labs

(b) New Zealand

(c) University OC-3

Fig. 3. Cache associativity

below 8%, shown with the dashed horizontal lines in the graphs.

Now let us look at the cost of increasing cache associativities. In our evaluation, we focus only on the storage costs of the cache. The additional costs of implementing associativity is not considered, since both the memory storage and associativity logic can be implemented in several different ways with varying costs. However, the results should be somewhat calibrated to the complexity that each additional level of associativity brings. For the example architecture given in Figure 2, the added costs of adding associativity scales with the level of associativity, not the number of entries. More specifically, the additional hardware for an N-way associative memory compared to direct-mapped memory is the logic gates that compare the output from each memory bank to the input FlowID, and the logic gates that select the right output. Approximately each additional FlowID comparison requires m XOR logic gates, where m is the FlowID size. An N-way selector requires $(N-1)$ AND gates and $(N-1)*k$ OR gates. Here, N is the cache associativity, and k is the result size. Thus, compared to a direct-mapped memory, the overall overhead of a N-way associative memory, in logic gates, is $(N-1)*m$ XOR, $(N-1)*k$ OR, and $(N-1)$ AND. If we assume the average memory cell size is approximately twice as big as a logic gate [19], the overhead of using N-way associative cache in memory cells, is

$$(N-1)*(m+k+1)/2 \tag{1}$$

bits.

With this derivation, we can estimate the overhead of increasing the cache associativities, in terms of silicon space usage. For example, for an IPv4 FlowID, the overhead of increasing the associativity from direct-mapped to 4-way is equivalent to using an additional silicon space of 21 bytes of memory, independent of the cache size. The real overhead of an N-way associative cache might be different because the derivation is based on the simplest implementation. However, as we can see early in this section, this overhead cost for 4-way associativities is very small compared to the improvement it makes to the overall cache performance.

We also evaluate the cache performance with higher than 4-way associative memory. The performance of higher associativities is between the result of 4-way associativities and the full associative cache, but is close to the performance of 4-way associative cache. Thus we conclude that a light associative memory, such as a 4-way associative one, is a good choice for the packet classification cache architecture.

B. Cache replacement

Cache replacement determines which entry, if any, must be replaced in order to make room for a newly classified flow. Intuitively, given the temporal locality of packets within most flows, an LRU strategy should provide the most effective results.

While such a strategy is sound when caches are relatively empty, it is not clear whether or not it is effective when

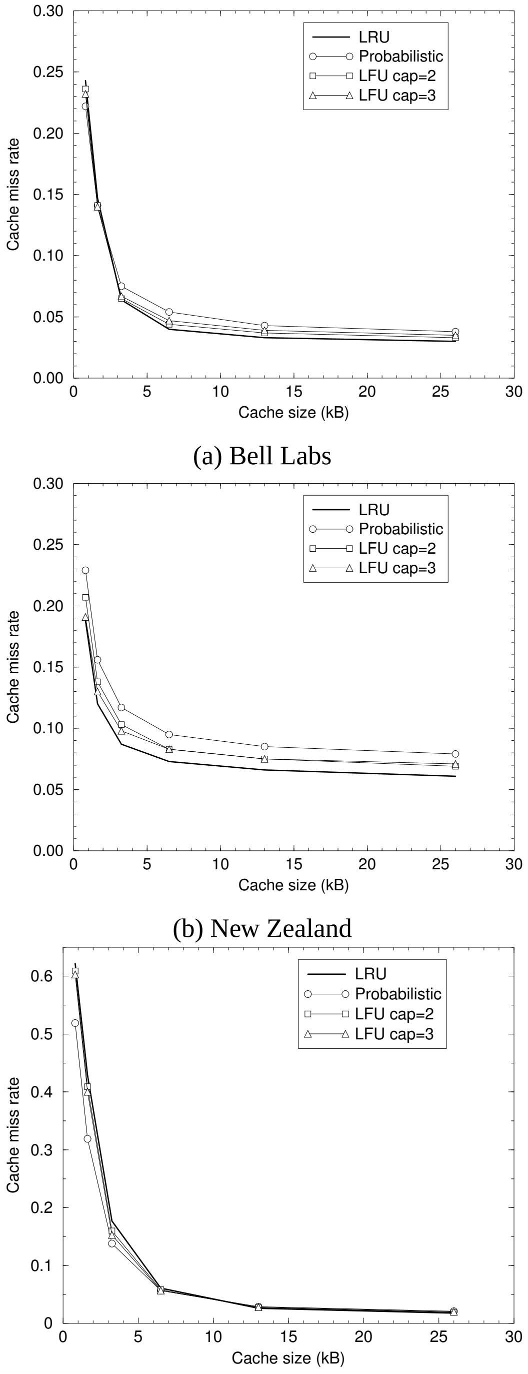
```
$alpha = 0.9$

Upon cache miss:
    $update(0)$
    Replace entry with probability $h$

Upon hit miss:
    $update(1)$

$update(how)$
    if (how == 0)
        $h$ = $alpha$*$h$
    else
        $h$ = $alpha$*$h$ + (1-$alpha$)
```

Fig. 4. Probabilistic replacement algorithm



(a) Bell Labs

(b) New Zealand

(c) University OC-3

Fig. 5. Replacement policies using 4-way caches

caches are more fully occupied. For example, consider the case of an on-line game flow consisting of a periodic stream of UDP packets [4] colliding in a direct-mapped cache with a short web transfer. Assuming the packets of the flows are perfectly interleaved, an LRU strategy exhibits a high degree of thrashing.

This thrashing phenomenon motivates our examination of a number of different cache replacement algorithms. In particular, we focus on a family of algorithms based on a Least-Frequently-Used (LFU) policy and a probabilistic replacement policy based on an estimate of the current miss rate. Both algorithms attempt to reduce thrashing by holding entries in the cache if they have a chance of producing a subsequent hit later on.

The LFU algorithm is fairly straight-forward. Each cache entry includes a small counter. Cache hits on an entry increment the entry's counter up to a certain cap limit, while cache misses decrement the counter. If a cache miss occurs and the counter is zero, the entry is replaced with the result of the new classification. In this paper, we examine counter caps of 2 and 3. Larger caps were also evaluated but they did not improve cache performance and are not included in this discussion.

The probabilistic algorithm is also fairly simple. The cache keeps a running estimate of its recent hit rate h. In our experiments, we use the exponentially weighted average of the hit rate as the estimate. When cache misses occur, the entry is replaced with probability h. Thus, when the hit rates plummet, the cache assumes that it is due to thrashing and attempts to hold onto entries longer in order to salvage some cache hits. Figure 4 shows the probabilistic algorithm used to avoid cache threshing.

Figure 5 shows the performance of the different cache replacement algorithms using a 4-way associative cache. Experiments with a range of different associativities were also

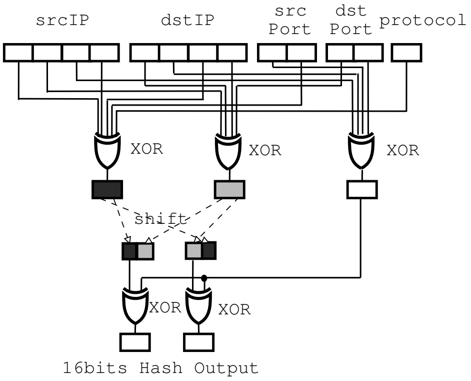
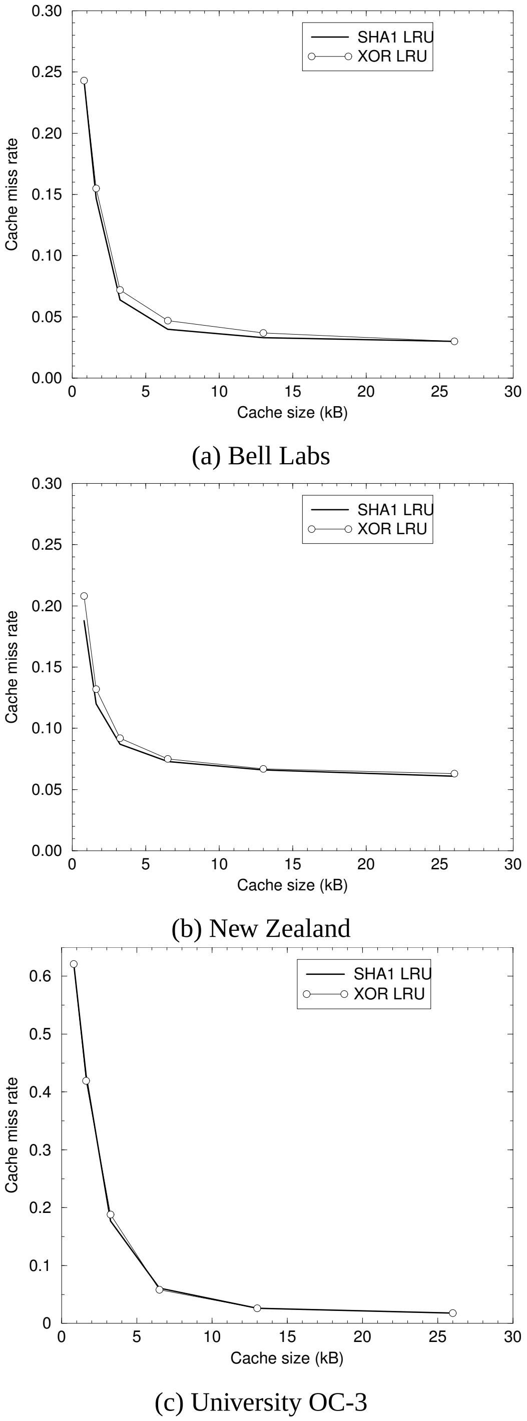


Fig. 6. XOR-based hash function

performed and showed similar results. Figure 5(c) shows that for the university trace that includes a large amount of gaming traffic (which is highly periodic in nature), replacement algorithms that attempt to prevent thrashing do have some impact on performance when the cache is underprovisioned. However, the benefit of using such replacement policies is insignificant compared to using LRU for the more traditional workloads as the cache becomes less fully occupied. In fact, for the BELL and New Zealand traces, both the LFU and the probabilistic algorithms increase miss rates due to stale flows being held in the cache for longer periods of time before being replaced. Thus, while LFU handles emerging continuous streams better when the cache is small, it should not be used in other circumstances.

C. Hash function

A critical component to implementing a cache is the hash function used to index into it. Without a strong hash function, a large number of collisions may occur leading to thrashing. Traditional hash functions, such as SHA-1 and MD5, are popular because they produce well-balanced output in which a single bit change in the input can change every bit of the output with equal probability. There are several shortcomings with using such hash functions in this context. While it would be preferable to use a hash function such as SHA-1 or MD5 on every packet header, the hardware costs for designing a low-latency, high-capacity hardware hash unit can be prohibitive, requiring a significant amount of logic circuitry. For example, according to the reference implementation [21], the generation of a SHA-1 hash output takes more than 1000 logic operations (Shift, AND, OR, and XOR) using 32-bit words. Building complex hash functions into hardware such as a network processor consumes precious transistors that could be otherwise used for other functions. Even worse, the delay caused by a hash function can be problematic since the delay of hardware implementations is proportional to the number of transistors on the signal path.



(a) Bell Labs

(b) New Zealand

(c) University OC-3

Fig. 7. Hash performance using 4-way, LRU caches

Recent work analyzing the mix of IP addresses present in real traffic [22] has shown that the allocation is highly structured. In particular, addresses tend to cluster around pre-CIDR address classes and often exhibit sequential allocation in the lower-order byte. Such address structure can be leveraged to potentially provide a faster, but slightly weaker hash function. To reduce the size of the hash function and the latency, we design a simple hash function based on XOR operations on the packet header that works on a small input size, and consumes only 16 logic operations (XOR and Shift). We show that a hash function as simple as ours can achieve similar performance result for implementing caches. Figure 6 shows the XOR-based hash algorithm.

In order to demonstrate the effectiveness of the weaker hash, Figure 7 compares the performance of our XOR-based hash versus the SHA-1 hash on a 4-way associative LRU cache. As the figure shows, the simple XOR-based hash function's performance is almost equal to that of the SHA-1 hash across all workloads.

While the XOR-based algorithm appears to be competitive, its impact on overall performance depends on several factors including the time it saves in execution and the penalty incurred by performing a full classification. For example, the speedup of using the XOR-based hash may be completely offset by the additional overhead in performing additional packet classifications due to the increasing miss rates.

VI. Conclusion

In this paper, we have presented results evaluating three important aspects of packet classification caching. Our experiments show that using memory with even a low level of associativity can improve the cache performance and significantly reduce the cache size. In addition, attention is needed to choose cache replacement policies for under-provisioned cache. LRU is generally the best replacement policies for most type of traffic. Finally, a fast, less complex hash can improve overall cache performance.

References

[1] I. Stoica, S. Shenker, and H. Zhang, "Core-Stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks," in *Proceedings of ACM SIGCOMM*, September 1998.

[2] K. Egevang and P. Francis, "IP Network Address Translator," *RFC 1641*, May 1994.

[3] Kimberly Claffy, *Internet traffic characterization*, Ph.D. thesis, San Diego, 1994.

[4] W. Feng, F. Chang, W. Feng, and J. Walpole, "Provisioning Online Games: A Traffic Analysis of a Busy Counter-Strike Server," in *Internet Measurement Workshop*, November 2002.

[5] Jun Xu, Mukesh Singhal, and Joanne Degroat, "A novel cache architecture to support layer-four packet classification at memory access speeds," in *INFOCOM*, 2000, pp. 1445–1454.

[6] T. V. Lakshman and Dimitrios Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM*, 1998, pp. 203–214.

[7] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, "Scalable high speed IP routing lookups," in *Proceedings of SIGCOMM '97*, September 1997, pp. 25–36.

[8] Pankaj Gupta and Nick McKeown, "Packet classification on multiple fields," in *SIGCOMM*, 1999, pp. 147–160.

[9] George Varghese, "Detecting packet patterns at high speeds," *SIGCOMM Tutorial*, August 2002.

[10] E. Johnson and A. Kunze, *IXP1200 Programming*, Intel Press, 2002.

[11] Nevil Brownlee and Margaret Murray, "Streams, flows and torrents," in *Passive and Active Measurement Workshop*, April 2001.

[12] S. McCreary and k. claffy, "Trends in wide area ip traffic patterns - a view from ames internet exchange," Tech. Rep., CAIDA, 2000.

[13] K.Thompson, G.Miller, and R.Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network*, 1997.

[14] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[15] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel, "Fast and scalable layer four switching," in *Proceedings of ACM SIGCOMM '98*, September 1998, pp. 191–202.

[16] C. Partridge, P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. Troxel, D. Waitzman, and S. Winterble, "A 50-Gb/s IP Router," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 237–245, June 1998.

[17] NLANR, "Passive Measurement and Analysis," `http://pma.nlanr.net/PMA/`.

[18] Anonymous, "Packet classification cache simulator," 2003.

[19] John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 2002.

[20] R. Jain, "Characteristics of destination address locality in computer networks: A comparison of caching schemes," *Computer Networks and ISDN Systems*, vol. 18, pp. 243–254, 1989/1990.

[21] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," *RFC 3174*, September 2001.

[22] E. Kohler, J. Li, V. Paxson, and S. Shenker, "Observed Structure of Addresses in IP Traffic," in *Internet Measurement Workshop*, November 2002.

[23] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers," in *25 Years ISCA: Retrospectives and Reprints*, 1998, pp. 388–397.