Approximate Caches for Packet Classification

Francis Chang, Wu-chang Feng
Systems Software Laboratory
OGI School of Science and Engineering at OHSU
Beaverton, Oregon, USA
{francis, wuchang}@cse.ogi.edu

Kang Li
Department of Computer Science
University of Georgia
Athens, Georgia, USA
kangli@acm.org

***Abstract*—Many network devices such as routers and firewalls employ caches to take advantage of temporal locality of packet headers in order to speed up packet processing decisions. Traditionally, cache designs trade off time and space with the goal of balancing the overall cost and performance of the device. In this paper, we examine another axis of the design space that has not been previously considered: accuracy. In particular, we quantify the benefits of relaxing the accuracy of the cache on the cost and performance of packet classification caches. Our cache design is based on the popular Bloom filter data structure. This paper provides a model for optimizing Bloom filters for this purpose, as well as extensions to the data structure to support graceful aging, bounded misclassification rates, and multiple binary predicates. Given this, we show that such caches can provide nearly an order of magnitude cost savings at the expense of misclassifying one billionth of packets for IPv6-based caches.**

Keywords— Bloom filter; packet classification; caches; probabilistic algorithms

I. INTRODUCTION

Modern network devices such as firewalls, network address translators, and edge routers rely on fast packet classification in order to perform well. These services require that packets be classified based on a set of rules that are applied to not only the destination address, but also flow identifiers such as source address, layer-4 protocol type, and port numbers. Unfortunately, packet classification is a very complex task. Because of this, there has been a large amount of work in developing more efficient classification algorithms [2][13][18][23][31][34]. Still, in the context of high-performance networks, the hardware requirements of performing full classification on each packet at line rates can be overwhelming[26].

To increase the performance of a packet classification engine, a cache is often employed to take advantage of temporal locality [8]. For example, caching has been shown to increase performance significantly with route lookups [22][35]. Caches are typically evaluated along two axes: size and performance. As additional storage is added, cache hit rates and performance increase. Unlike route caches that only need to store destination address information, packet classification caches require the storage of full packet headers. Unfortunately, due to the increasing size of packet headers (the eventual deployment of IPv6 [19]), storing full header information can be prohibitive given the cost of the high-speed memory that would be used implement such a cache. To address this problem, this paper examines a third axis for designing packet classification caches: accuracy. In particular, we seek to answer the following question:

What are the quantifiable benefits that relaxing the accuracy of a cache has on the size and performance of packet classification caches?

While there are many ways of exploring this axis, this paper examines one approach for doing so through the use of a modified Bloom Filter. In this approach, classified packets satisfying a binary predicate are inserted into the filter that caches the decision. Subsequent packets then query the filter to quickly test membership before being processed further. Packets that hit in the filter are processed immediately, based on the predicate, while packets that miss go through the full packet classification lookup process.

In this paper, we briefly describe the Bloom filter and analyze its properties. In particular, we examine the exact relationship between the size and dimension of the filter, the number of flows that can be supported, and the misclassification probability incurred. While Bloom filters are good for storing binary set membership information, realistic network devices classify packets into many, possibly disjoint, sets. To address this limitation, we extend the basic approach to support multiple binary predicates and analyze its expected performance. Another issue in using Bloom filters in such a manner is the highly dynamic nature of the "dictionary" of packet headers. In particular, such a cache must be able to evict stale entries and preserve a bounded maximum misclassification rate. To address these issues, we present the design and evaluation of extensions for gracefully aging the cache over time to minimize misclassification. We also explore the design and implementation of such a cache on a modern network processor platform.

Section II covers related work and Section III introduces Bloom filters. Section IV extends the Bloom filters to support multiple binary predicates while Section V analyzes

This work supported by the National Science Foundation under Grant EIA-0130344 and the generous donations of Intel Corporation. Any opinions, findings, or recommendations expressed are those of the author(s) and do not necessarily reflect the views of NSF or Intel.

extensions for gracefully aging the cache. Section VI explores the performance impact of running a Bloom filter on a network processor, while Section VII discusses the potential results of misclassification.

II. Related Work

Due to the high processing costs of packet classification, network appliance designers have resorted to using caches to speed up packet processing time. Early work in network cache design borrowed concepts from computer architecture (LRU stacks, set-associative multi-level caches) [6][27]. Some caching strategies rely on CPU L1 and L2 cache [22] while others attempt to map the IP address space to memory address space to use the hardware TLB [26]. Another approach is to add an explicit timeout to an LRU set-associative cache to improve performance by reducing thrashing [35]. More recently, in addition to the leveraging the temporal locality observed on networks, approaches to improving cache performance have applied techniques to compress and cache IP ranges to take advantage of the spatial locality in the address space of flow identifiers [7][17]. This effectively allows multiple flows to be cached to a single cache entry, so that the entire cache may be placed into small high-speed memory such as a processor's L1/L2 cache.

Much of this work is not applicable to layer-4 flow identification that is the motivation for our work. Additionally, all of these bodies of work are fundamentally different from the material presented in this paper, because they only consider exact caching strategies. Our approach attempts to balance performance and resource requirements with an allowable error rate.

Recently, there have been proposed ideas in the area of applying Bloom Filters as an acceleration technique to route-lookup functions [11]. This technique uses Bloom Filters to direct hash probes to main memory. In this case, a matched Bloom filter signature implies the possibility that a hash probe to main memory may be successful, but does not encode any information itself.

III. Theory

We use a Bloom-filter to construct our approximate cache. A Bloom filter is a space-efficient data structure designed to store and query set-membership information [1].

Bloom filters were originally invented to store large amounts of static data (for example, hyphenation rules on English words). In recent years, this data structure has been rediscovered by the networking community, and has become a key component in many networking systems [3][25][33].

Applications of Bloom filters in computer networking include web caching [12], active queue management [14], IP traceback [29][30], and resource routing [4][9].

A. The Bloom Filter

In our implementation, a Bloom filter data structure consists of $M = N \times L$ bins. (Each bin consists of one bit.) These bins are organized into L levels with N bins in each level, to create N^L virtual bins (possible permutations). To interact with the Bloom filter, we maintain independent hash functions, each associated with one bin level. Each hash function maps an element into one of the N bins in that level. For each element of the set, $S = \{e_1, e_2, \ldots e_k,\}$, we compute the L hash functions, and set all of the corresponding bins to 1. To test membership of any element in our Bloom filter, we compute the L hash functions, and test if all of the corresponding buckets are set to 1. See Figure 1 for an example.

This approach may generate false positives – a Bloom filter may incorrectly report that an element is a member of the set S – but a Bloom filter will never generate false negatives.

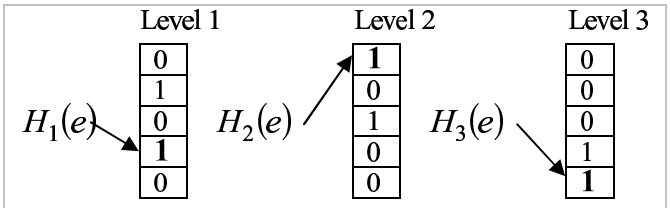

Figure 1: An example: A Bloom filter with $N = 5$ bins and $L = 3$ hash levels. Suppose we wish to insert an element, e.

For optimal performance, each of the L hash functions, $H_1, H_2, \ldots, H_L$, should be a member of the class of universal hash functions [5]. That is, each hash function should distribute elements evenly over the hash's address space, and for each hash function $H_\varepsilon : e \to [1 \ldots N]$, the probability of 2 distinct elements colliding is $1/N$. That is to say, $P(H_\varepsilon(a) = H_\varepsilon(b), a \neq b) = 1/N$.

In practice, we apply only one hash function, $H : e \to [1 \ldots N^L]$, for each insertion or query operation, and simply use different portions of the resulting hash to implement the L hash functions.

Our definition of a Bloom filter differs slightly from the original definition [1], where each of the L hash functions can address all of the M bit buckets. This definition of the Bloom filter is often used in current designs due to potential parallelization gains to be had by artificially partitioning memory [14]. It should be noted that this approach yields a slightly worse probability of false positives under the same conditions, but an equal asymptotic false-positive rate [3].

B. Properties of the Bloom Filter

In order to better design our cache and understand its limitations, it is important to understand the behavioural properties of a Bloom filter. In particular, we are interested in how the misclassification probability, and the size of the Bloom filter, will affect the number of elements that it can store.

Let us take the example of a firewall to motivate our analysis. The rationale of a firewall is to restrict and censor traffic between the internal and external networks. A firewall acts as both an entry point into and exit point from the network. As such, it must be able to process all traffic travelling to and from a network at line speed. This makes it a

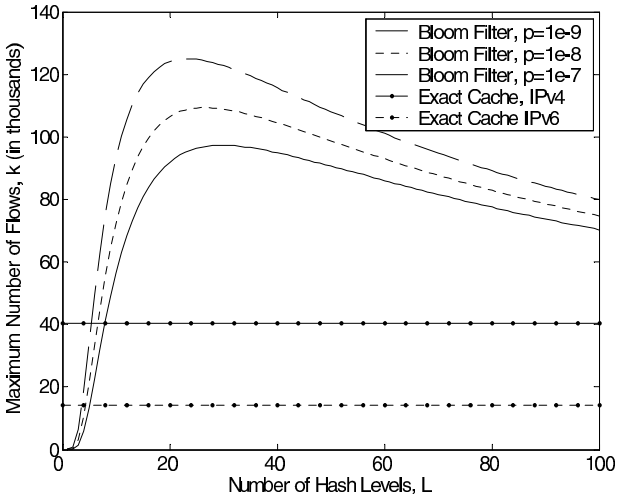

Figure 2: The maximum number of flows that can be stored by a 512KB cache

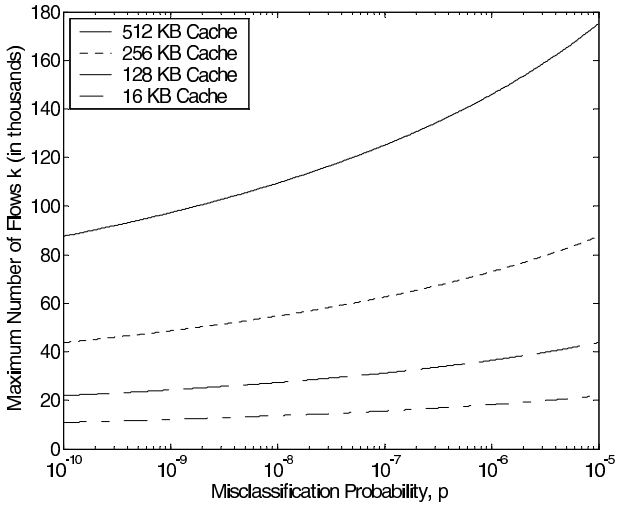

Figure 3: The trade-off between the misclassification probability, p, and the maximum number of elements (flows), k, using optimum values of L.

simple example in which to apply our approximate cache. Allowed flows are inserted into the cache, while new and censored flows are not.

A Bloom filter with N buckets in each of its L levels, storing k elements has a probability of yielding a false positive of

$$p = \left(1 - \left(1 - \frac{1}{N}\right)^{k}\right)^{L}$$

For our purposes, we need to know how many elements (flows), k, we can store in our bloom filter, without exceeding some misclassification probability, p. Solving for k, we get

$$k = \frac{\ln(1 - p^{1/L})}{\ln(1 - 1/N)}$$

To simplify this equation, we apply the approximation $1 - 1/N \approx e^{-1/N}$. So, we construct $\kappa \approx k$,

$$\begin{aligned}\kappa &= \frac{\ln(1 - p^{1/L})}{\ln(e^{-1/N})} \\ &= -N \ln(1 - p^{1/L}) \\ &= -\frac{M}{L} \ln(1 - p^{1/L})\end{aligned}$$

From this equation, it is clear that the number of elements, κ, a Bloom filter can support scales linearly with the amount of memory M. The relative error of this approximation, κ/k, grows linearly with the number of hash functions L, and decreases with increasing M. For the purposes of our application of this approximation, the relative error is negligible[1].

[1] For $M \geq 1024$ bytes, and $L \leq 50$, the relative error is less than 0.35%.

Note that solving for p in this equation yields the more popular expression [3][12][30],

$$p = \left(1 - e^{-\kappa L/M}\right)^{L}$$

C. *Dimensioning a Bloom Filter*

Bloom-filter design was originally motivated by the need to store dictionaries in memory. The underlying design assumption is that the data is static. However, this assumption no longer holds when dealing with network traffic. Previous work has often attempted to dimension a Bloom filter such that the misclassification rate is minimized for a fixed number of elements [3].

To apply Bloom filters to the problem of storing a cache, we prefer to maximize the number of elements, k, that a Bloom filter can store, without exceeding a fixed maximum tolerable misclassification rate, p.

To maximize κ as a function of L, we take the derivative, $d\kappa/dL$, set it to 0, and solve for L to find the local maximum.

$$\frac{d\kappa}{dL} = \frac{d}{dL}\left(-\frac{M}{L}\ln(1 - p^{1/L})\right)$$

$$\frac{d\kappa}{dL} = -\frac{M}{L^2}\left(\ln(1 - p^{1/L}) - \frac{p^{1/L}\ln p}{L(1 - p^{1/L})}\right)$$

$$0 = \frac{M}{L^2}\left(\ln(1 - p^{1/L}) - \frac{p^{1/L}\ln p}{L(1 - p^{1/L})}\right)$$

$$0 = \ln(1 - p^{1/L}) - \frac{p^{1/L}\ln p}{L(1 - p^{1/L})}$$

So, now suppose a $u = p^{1/L}$, so $L = \ln p / \ln u$. Then,

$$0 = \ln(1 - u) - \frac{u \ln p}{(\ln p / \ln u)(1 - u)}$$

$$0 = \ln(1-u) - \frac{u \ln u}{(1-u)}$$

$$u \ln u = (1-u)\ln(1-u)$$

$$u^u = (1-u)^{1-u}$$

Since $p \in [0,1]$, then $u \in [0,1]$, so u only has one solution, $u = ½$, which means κ is maximized for

$$L = -\ln p / \ln 2 = -\log_2 p$$

This is an interesting result, because it implies that L is invariant with respect to the size of the Bloom filter, M.

Another interesting implication of this equation is that the Bloom filter is "optimally full" when half of all the buckets are set ($p = \frac{1}{2}^L$).

It should be noted that the accuracy of this approximation ($\kappa \approx k$) increases with M. In our testing, for cache sizes greater than 1KB, this approximation yields no error. In all the simulations presented in this paper, this approximation and the optimal value of L are equal. Even if we choose a slightly sub-optimal value of L, the difference in the maximum number of flows the Bloom filter can store is negligible (Figure 2). For comparison, we will reintroduce the concept of an exact cache – the traditional cache that does not yield false positives or negatives.

A less obvious implication of this approximation is the relationship between the amount of memory, *M*, the number of elements (flows), *k*, and the probability of a false positive, p.

Figure 3 graphs the relationship between *p* and *k*. We can see that the relationship is roughly logarithmic. This approximation serves as a good guide for ranges of two orders of magnitude or less.

Since the optimal choice of L is asymptotically invariant with respect to M, and κ is proportional to M, we can assert that k is linearly related to M. A visual representation of this relationship is depicted in Figure 4. Note that a Bloom filter cache with a misclassification rate of one in a billion can store more than twice as many flows as an exact IPv4 cache, and almost 8 times as much as an exact IPv6 cache. (Each entry in an exact IPv6 (37 bytes) cache consumes almost 3 times as much memory as an IPv4 entry (13 bytes) [19].)

It is also important to note that, with our scheme, it is possible to store mixed IPv4/IPv6 traffic without making any major changes to our design.

To summarize:

- The optimal value of L, the number of levels, is invariant with respect to the size of the Bloom filter, M.
- The number of elements, k, and the misclassification probability, p, are roughly logarithmically related.
- k is linearly related to M.
- An optimally full Bloom filter has ½ of its bits set.

IV. MULTIPLE PREDICATES

Our first extension to the Bloom filter is to extend its storage capability to support multiple binary predicates (as opposed to the single binary predicate yes/no data storage of a traditional Bloom filter). This extension is needed for more sophisticated applications, such as routers, which need to record forwarding interfaces.

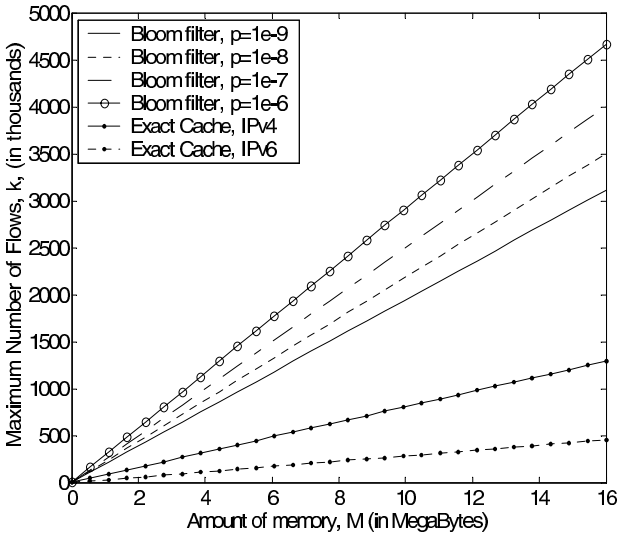

Figure 4: The relationship between the amount of memory, M, and the maximum number of elements, k (Bloom M Filters use an optimum value of L)

We propose a modification to our existing algorithm that allows us to store multiple binary predicates, while preserving the desired original operating characteristics of the Bloom filter cache.

Consider a router with I interfaces. The cache would be required to store a routing interface number. To support this, a data structure that can record I binary predicates is required. To store this information, we will construct a cache composed of I Bloom filters.

Suppose we are caching a flow, e, that should be routed to the i^{th} interface. We would simply insert e into the i^{th} Bloom filter in our cache. This encoding scheme is similar to "1-hot" encoding.

To query the cache for the forwarding interface number of flow e, we will simply need to query all I Bloom filters. If e is a member of the i^{th} Bloom filter, this implies that flow e should be forwarded through the i^{th} interface.

If e is not a member of any Bloom filter, e has not been cached. In the unlikely event that more than one Bloom filter claims e as a member, we have a confounding result. One solution to this problem is to treat the cache lookup as a miss by reclassifying e. This approach preserves correctness while adding only minimal operating overhead.

The probability of misclassification, p, with this algorithm is

$$p = 1 - \left(1 - \left[1 - (1 - 1/N')^{k'}\right]^{L'}\right)^{I}$$

Solving for k', the maximum number of flows this approach can store, we find

$$k' = \frac{\ln\left(1-\left[1-(1-p)^{1/I}\right]^{1/L'}\right)}{\ln(1-1/N')}$$

Using the same technique discussed earlier in Section 4.2, we find that k' is maximized when

$$L' = -\frac{\ln\left(1-(1-p)^{1/I}\right)}{\ln(2)}$$

The proposed extension to the Bloom filter cache requires increasing the number of memory accesses by a factor of I. Additional memory accesses can incur a serious performance penalty. Taking advantage of the memory bus width can easily mitigate this disadvantage by the following technique:

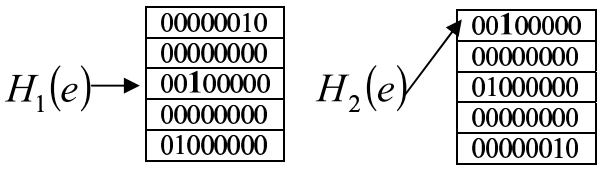

Figure 5: An example: A modified Bloom filter with 5 buckets and 2 hash levels, supporting a router with 8 interfaces. Suppose we wish to cache a flow e that gets routed to interface number 2.

Consider a Bloom filter in which each bucket can store a pattern of I bits, where bit i represents interface i. When adding a packet to the bloom filter, we would only update bit i of each bucket.

When querying the modified Bloom filter for a flow, e, we will take the results from each level of the bloom filter, and ***AND*** the results. An example is depicted in Figure 5.

Thus, a router, with I Bloom filters, each Bloom filter having L hash levels, need only make L memory accesses to insert or query the cache.

A. Non-Uniform Distributions

The equations presented earlier in Section 5 assume that elements are evenly distributed over the multiple binary predicates. If the elements are not evenly distributed, our modified Bloom filter can become polluted in a short amount of time.

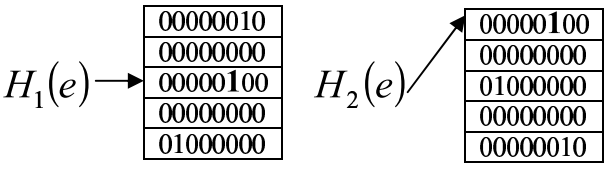

Figure 6: As before, suppose flow e is to be forwarded to interface 2. Now, let us suppose that $H'(e)=3$. So, $j=(i+H'(e)) \bmod I=(2+3) \bmod 8=5$.

For example, consider a router with 16 interfaces (binary predicates), using 1KB of memory and a misclassification probability, p, of 1e-9. If flows are distributed evenly over the interfaces, this configuration can support 167 elements. Conversely, if 90% of flows set the first predicate, it would require only 13 elements to "fill" this Bloom filter.

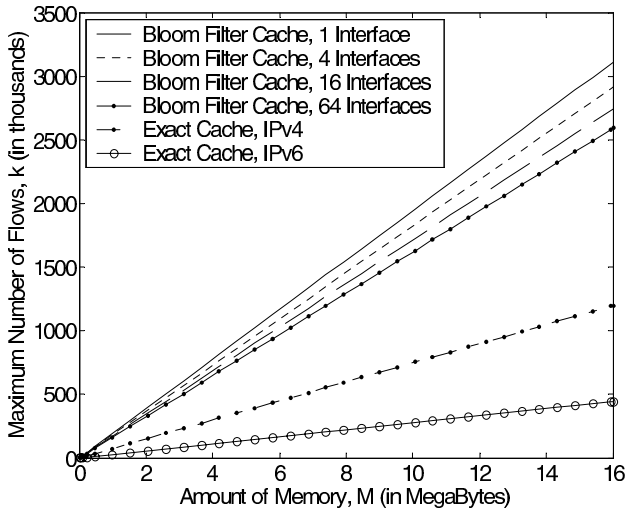

Figure 7: Effect of storing routing information on effective cache size, $p=1e-9$, using optimal Bloom filter dimensions

To compensate for this deficiency, consider a new hashing function, $H': e \rightarrow [0 \ldots I-1]$, and let $j=(i+H'(e)) \bmod I$. Instead of setting bit i in a Bloom filter, we will set bit j. (See example in Figure 6.)

This approach ensures that set bits are uniformly distributed throughout the cache, even when the elements are not evenly distributed.

B. Multi-Predicate Comparison

It is important to examine how the multiple binary-predicates Bloom cache compares to the single-predicate case to understand how our extension affects the behaviour of our Bloom filter cache.

As discussed previously, the single-bit Bloom filter cache can store a maximum of $\kappa = -M/L \ln(1-p^{1/L})$. For an optimized choice of $L=-\ln p / \ln 2$, κ becomes

$$\kappa_{\max} = \frac{M \ln(2)}{\ln(p)} \ln(1-p^{-\ln 2/\ln p})$$

The maximum number of flows the modified multi-bit Bloom filter can store is

$$k' = \frac{\ln\left(1-\left[1-(1-p)^{1/I}\right]^{1/L'}\right)}{\ln\left(1-L'/M\right)}$$

Applying the approximation $1-1/N \approx e^{-1/N}$ we find

$$\kappa' = -\frac{M}{L'} \ln\left(1-\left[1-(1-p)^{1/I}\right]^{1/L'}\right)$$

When L' is optimized, κ' becomes

$$\kappa'_{\max} = \frac{M \ln(2)}{\ln\left(1-(1-p)^{1/I}\right)} \ln\left(1-\left[1-(1-p)^{1/I}\right]^{p}\right)$$

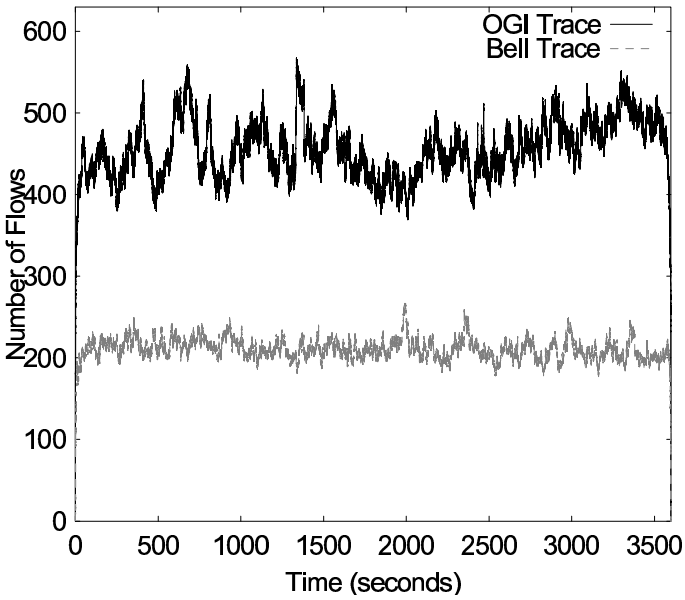

Figure 8: Number of concurrent flows in test data sets

where

$$v = \frac{-\ln(2)}{\ln\left(1-(1-p)^{1/I}\right)}$$

Immediately, we can see that the two approaches are still linearly related in M. (Note that I and p are constants.) This is an important property, because it means that our proposed algorithm preserves the behaviour of the single binary predicate cache.

Figure 7 compares the difference in the maximum number of flows that can be stored by a multi-bit Bloom filter cache.

To better determine the relative performance of the multiple binary predicate and the single-binary-predicate cache approaches, we take the difference in the maximum number of flows that each design will accommodate.

The difference of the two approaches is,

$$\kappa_{\max} - \kappa'_{\max} = M(\ln 2)^2\left(\frac{1}{\ln\left(1-(1-p)^{1/I}\right)} - \frac{1}{\ln(p)}\right)$$

For $p \ll 1$, $(1-p)^{1/I} \cong 1 - p/I$, giving

$$\kappa_{\max} - \kappa'_{\max} \cong \frac{M(\ln 2)^2}{\ln p}\left(\frac{\ln I}{\ln p - \ln I}\right)$$

If I is not very big, as is the case when considering the number of interfaces of a router (for reference, a Juniper T640 routing node has 160 interfaces) then $-\ln p >> \ln I$, we can approximate by

$$\kappa_{\max} - \kappa'_{\max} \cong \frac{M(\ln 2)^2}{\ln p}\left(\frac{\ln I}{\ln p - 0}\right) = \frac{M(\ln 2)^2 \ln I}{(\ln p)^2}$$

This is an overestimate of the difference. So, we can say that, at worst, this approach scales logarithmically with I (for M and p constant).

It is surprising how effective this approach is (Figure 7). The algorithm does not pollute the Bloom Filter (setting more bits) than the single binary-predicate approach. However, it is slightly more susceptible to pollution (each membership query examines $L \times I$ bits, as opposed to the L bits of the single binary predicate Bloom filter).

It should be noted that the multi-predicate solution is a superset of the single-predicate solution – setting I to 1 yields the equations presented in Section 4.1.

V. BLOOM FILTER AGING

Our second extension to the Bloom filter is adding the ability to evict stale entries from the cache. Bloom filters were originally designed to store set membership information of unchanging, or expanding sets. We must adapt this algorithm to allow graceful eviction of elements to use this data structure effectively in a dynamic environment such as the Internet.

The first step towards developing an algorithm to age a Bloom filter is to decide how much information has already been stored in the cache. A simple method of deciding when the cache is full is to choose a maximum tolerable misclassification probability, p. Whenever the instantaneous misclassification probability exceeds this constant, $(p_{instantaneous} > p)$, we consider the Bloom filter to be "full."

We can calculate $p_{instantaneous}$ by using different means. Let $\omega_1, \omega_2, \ldots, \omega_L$ be the fractions of buckets in each level of the Bloom filter that are set. The probability of misclassification is simply the product of ω_i's.

$$P_{misclassifcation} = \prod_{i=1}^{L} \omega_i$$

This method will accurately estimate the misclassification probability. The drawback to this approach is that it will require counting the exact number of bits we set, complicating later parallel access implementations of this algorithm, as well as adding several per-packet floating-point operations.

We can devise a simpler estimate of $P_{misclassification}$ that does not involve precise bit counting nor global synchronization, by applying knowledge of the properties of the Bloom filter discussed earlier. We need simply to count the number of flows k' that we insert into our Bloom filter. So our estimate of the misclassification probability becomes

$$P_{misclassification} = \left[1-(1-1/N)^{\kappa'}\right]^L$$

Reversing this equation, and solving for k_{max} we get

$$k_{max} = \left\lfloor \log\left(1-P_{max}^{1/L}\right) / \log(1-1/N) \right\rfloor$$

This estimate also provides the benefit of simplicity of calculation – floating-point arithmetic is no longer required during runtime, only an integer comparison $(k' > k_{max})$. Additionally, it becomes easier to gauge the behaviour of the cache - k' increases linearly with the number of new flows we observe.

Now, let us turn to the problem of applying this information to age the Bloom filter cache.

A. Cold Cache

This naïve approach to the problem of Bloom filter aging involves simply emptying the cache whenever the Bloom filter becomes "full."

The main advantage to this solution is that it makes full use of all of the memory devoted to the cache, as well as offering a simple implementation while maintaining a fixed worst-case misclassification probability.

The disadvantages, however, are quite drastic when considering the context of a high-performance cache:

- While the cache is being emptied, it cannot be used.
- Immediately after the cache is emptied, all previously cached flows must be re-classified, causing a load spike in the classification engine.
- Zeroing out the cache may cause a high amount of memory access.

This approach mainly serves as a reference point to benchmark further algorithm refinement.

B. Double-Buffering

If we partition the memory devoted to the cache into two Bloom filters, an active cache and a warm-up cache, we can more gracefully age our cache. This approach is similar to the one applied in Stochastic Fair Blue [14]. The basic algorithm is as follows:

when a new packet arrives
  if the flow id is in the active cache
    if the active cache is more than ½ full
      insert the flow id into the warm-up cache
    allow packet to proceed
  otherwise
    perform a full classification
    if the classifier allows the packet
      insert the flow id into the active cache
      if the active cache is more than ½ full
        insert the flow id into the warm-up cache
      allow packet to proceed
if the active cache is full
  switch the active cache and warm-up cache
  zero out the old active cache

The goal of this approach is to avoid the high number of cache misses immediately following cache cleaning. By switching to a background cache, we can start from a "warmed-up" state. This approach can be thought of as an extremely rough approximation of LRU.

However, this approach also has its drawbacks:

- Double the memory requirement to store the same number of concurrent flows, as compared to the cold-cache case.
- Zeroing out the expired cache still causes a load spike in the use of the memory bus (although it is a smaller spike). This can be partially mitigated by slowly zeroing out memory.
- In the simplest implementation, this algorithm can potentially double the number of memory accesses required to store a new flow. This performance loss can be recovered by memory aligning the two bloom filters, so that fetching a word of memory will return the bit states of both Bloom filters.

Now, let us turn to the problem of applying this information to age the Bloom filter cache.

C. Evaluation

For evaluation purposes, we used two datasets, each of one hour in length. The first of the datasets was collected by Bell

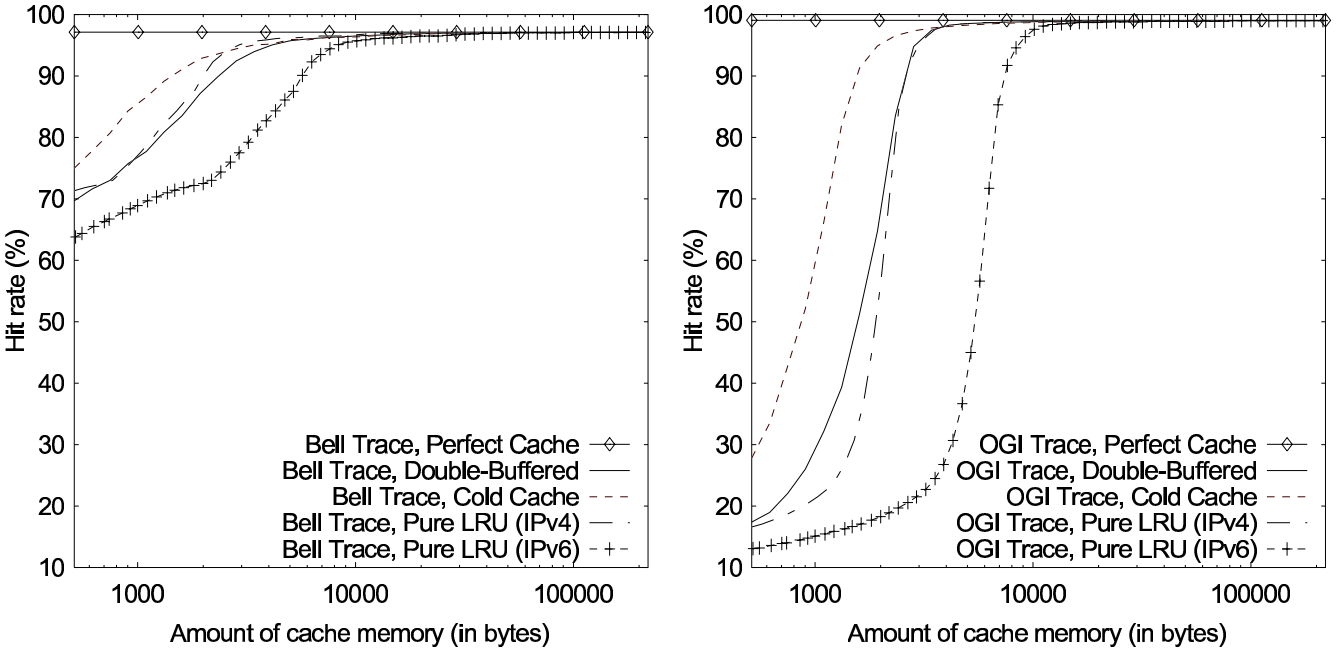

Figure 9: Cache hit rates as a function of memory, *M*

Labs research, Murray Hill, NJ, at the end of May 2002. This dataset was made available through a joint project between NLANR PMA and Internet Traffic Research Group [28]. The trace was of a 9 Mb/s Internet link, serving a staff of 400 people.

The second trace was a non-anonymized trace collected at our local university OC-3c link. Our link connects with Internet2 in partnership with the Portland Research and Education Network (PREN). This trace was collected on the afternoon of July 26th, 2002.

	Bell Trace	OGI Trace
Trace Length (seconds)	3600	3600
Number of Packets	974613	15607297
UDP Packets	671471	10572965
TCP Packets	303142	5034332
Number of Flows	32507	160087
Number of TCP Flows	30337	82673
Number of UDP Flows	2170	77414
Avg. Flow Length (seconds)	3.2654	10.2072
Avg. TCP Flow Length (seconds)	13.8395	11.2555
Avg. UDP Flow Length (seconds)	155.0410	9.0877
Longest Flow (seconds)	3599.95	3600
Avg. Packets/Flow	29.9816	97.4926
Avg. Packets/TCP Flow	9.9925	60.8945
Avg. Packets/UDP Flow	309.434	136.577
Max # of Concurrent Flows	268	567

Table 1: Sample Trace Statistics

Table 1 presents a summary of the statistics of these two datasets. A graph of the number of concurrent flows is shown in Figure 8.

For the purposes of our analysis, a bi-directional flow is considered as 2 independent flows. A flow begins when the first packet bearing a unique 5-tuple (source IP address, destination IP address, protocol, source port, destination port) arrives at our node. A flow ends when the last packet is observed, or after a 60 second timeout. The timeout is chosen in accordance with other measurement studies [16], and observations in the field [22][24].

	Bell Trace	OGI Trace
Hit Rate	0.9714	0.9877
Maximum misses (over 100ms intervals)	6	189
Variance of misses (over 100 ms intervals)	1.35403	17.4375
Average misses (over 100 ms intervals)	0.7749	5.8434

Table 2: The results of a perfect cache Sample trace statistics

As a reference, we introduce the idea of a perfect cache – a fully associative cache, with an infinite amount of memory. This cache only takes compulsory cache misses (the theoretical minimum). The fundamental performance statistics are reported in Table 2.

For a comparison with exact caching schemes, we simulate a fully associative cache using an LRU replacement policy. The performance of this scheme is presented in Figure 9. LRU was chosen because of its near-optimal caching performance in networking contexts [22][35].

This simulation is intended to represent best-case exact caching performance, even though it is infeasible to implement a fully associative cache on this scale.

For our simulation, we use the SHA1 hash function [15]. It should be noted that the cryptographic strength of the SHA1 hash does not increase the effectiveness of our implementation. It is important to recognize that other, faster hashing algorithms exist. Using a hardware-based hashing implementation is also possible. In the IXP1200 [21], the hardware hash unit can complete a hashing operation every nine clock cycles.

For the purposes of this study, we use a misclassification probability of one in a billion. Typically, TCP checksums will fail for approximately 1 in 1100 to 1 in 32000 packets, even when link-level CRCs should only admit error rates of 1 in 4 billion errors. On average, between 1 in 16 million to 1 in 10 billion TCP packets will contain an undetectable error [32]. We contend that an imprecision of this magnitude will not meaningfully degrade network reliability.

To support an exact IPv4 cache, a 4-way set associative IPv4 cache requires a 52-byte memory read on each cache lookup. A 4-way associative IPv6 cache would require a 148-byte memory access. To support an approximate cache with an error rate of 1 in a billion would require 30 1-bit memory fetches.

D. Cold Cache Performance

With the Bell dataset, the cold cache performs reasonably, using 4 KB of cache memory, and a misclassification probability of 1e-9. The optimal dimensioning for Bloom filters of this size is 30 hash functions, storing a maximum of 611 concurrent flows.

Throughout the 1-hour trace, there were no misclassifications and an overall cache hit-rate of 95.15%. Aggregated over 100ms intervals, there were a maximum of 8 cache misses/100ms, with an average of 1.32 and a variance of 10.33.

Figure 10 illustrates the cache misses during a portion of the trace. We can see that emptying the cache corresponds to a spike in the amount of cache misses that is not present when using a perfect cache. This spike is proportional to the number of concurrent flows. This type of behaviour will apply undue pressure to the classification engine, resulting in overall performance degradation.

E. Double-Buffering Performance

Using a double-buffered approach can smooth the spikes in cache misses associated with suddenly emptying the cache.

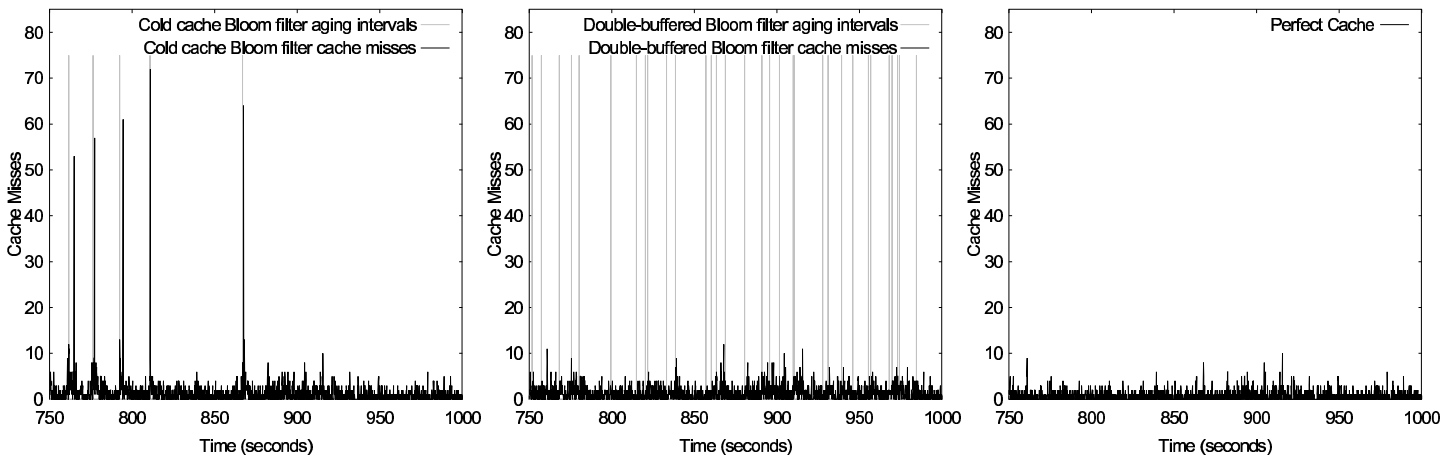

Figure 10: Comparing cold cache and double-buffered bloom caches using 4 KB of memory (Bell dataset)
("Aging intervals" represents transition points in time only, and does not represent any "vertical magnitude")

Double-buffering effectively halves the amount of immediately addressable memory, in exchange for a smoother aging function. As a result, this bloom filter was only able to store 305 flows for a 4096 byte cache, in comparison with the 611 flows of the cold-cache implementation.

This implementation had a slightly lower hit rate of 95.04% with the Bell dataset. However, we succeeded in reducing the variance to 5.43 while maintaining an average cache miss rate of 1.34/100ms. Viewing Figure 10, we can see that the correspondence between cache aging states and miss rates does not correspond to performance spikes as prevalently as in the cold cache implementation.

This implies that the double-buffered approach is an effective approach to smoothing out the performance spikes present in the cold cache algorithm. To better quantify the "smoothness" of the cache miss rate, we graph the variance, and average miss rates (Figure 11 and Figure 12).

From Figure 9 and Figure 11, we observe that for a memory-starved system, the cold-cache approach is more effective with respect to cache hit-rates. It is surprising how effective this naïve caching strategy is, with respect to overall cache performance. Moreover, we note that it performs better than both an IPv6, and IPv4 exact cache, with both datasets for a memory poor cache, and keeps pace as memory improves. As the amount of memory increases, we can see that the double-buffered approach is slightly more effective in reducing the number of cache misses.

Looking to Figure 12, we observe that the variance in miss rates decreases much faster in the double-buffered case than in the cold-cache approach. It is interesting to note that in the OGI trace, the variance actually increases, before it decreases. Interpreting Figure 11 and Figure 12, we can see that for a very memory-starved system, the variance is low because the cache miss rate is uniformly terrible.

Comparing the double-buffered approximate cache implementation to exact caching gives comparable performance when considering an IPv4 exact cache, even though the approximate approach can cache many more flows. This is due to the imprecision of the aging algorithm – an LRU replacement policy can evict individual flows for replacement, whereas a double-buffered approach must evict ½ of cached flows at a time. However, when considering IPv6 data structures, this disadvantage is overshadowed by the pure amount of storage capacity a Bloom filter can draw upon.

It is important to note that in all of these graphs, the behaviour of each of the systems approaches optimum as memory increases. This implies that our algorithm is correct and does not suffer fundamental design issues.

VI. Hardware Overhead

A preliminary implementation on Intel's IXP1200 Network Processor [21] was constructed, to estimate the amount of processing overhead a Bloom filter would add.

The hardware tested was an IXP1200 board, with a 200 MHz StrongARM, 6 packet-processing microengines and 16 Ethernet ports.

A simple micro-engine level layer-3 forwarder was implemented as a baseline measurement. A Bloom filter cache implementation was then grafted onto the layer-3 forwarder code base. A null classifier was used, so that we could isolate the overhead associated with the Bloom filter lookup function. No cache aging strategy was used. The cache was placed into SRAM, because scratchpad memory does not have a pipelined memory access queue, and the SDRAM interface does not support atomic bit-set operations

A. IXP Overhead

The performance of our implementation was evaluated on a simulated IXP1200 system, with 16 virtual ports. The implementation's input buffers were kept constantly filled, and we monitored the average throughput of the system.

The Bloom filter cache implementation was constructed in a way to ensure that no flow identifier was successfully matched, and each packet required an insertion of its flow ID into the Bloom filter. This was done so that the worst possible performance of a Bloom filter cache could be ascertained. The code was structured in a way to disallow any shortcutting or

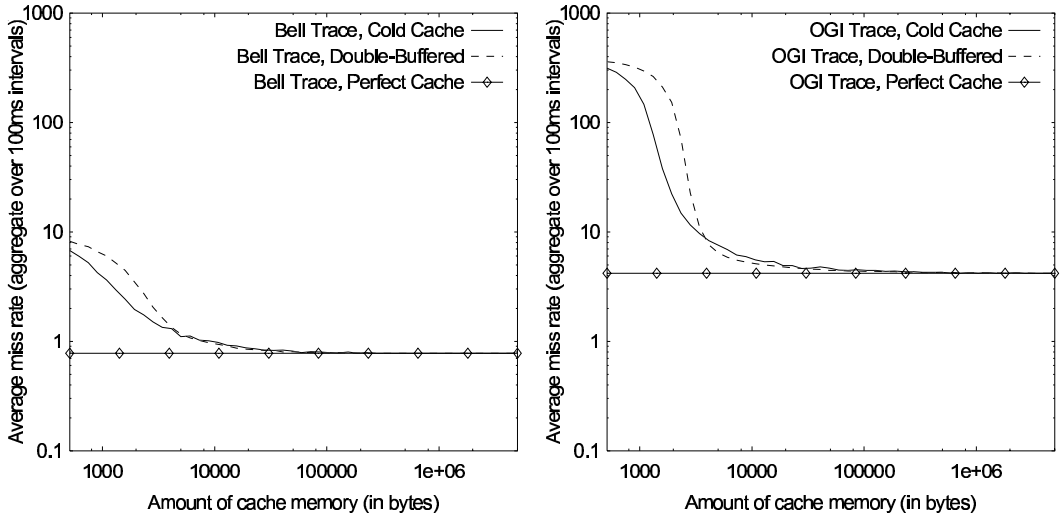

Figure 11: Average cache misses as a function of memory, *M* (aggregate over 100ms timescales)

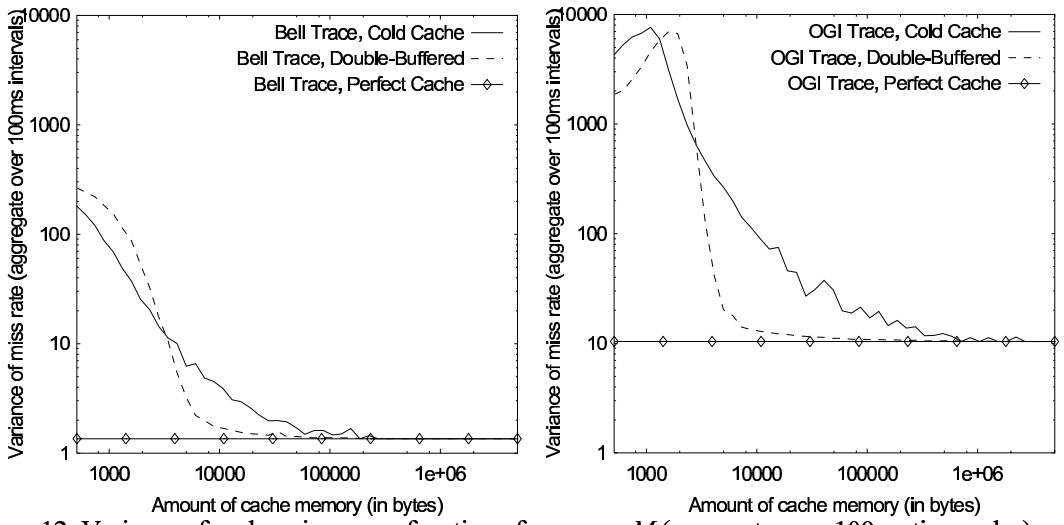

Figure 12: Variance of cache misses as a function of memory, *M* (aggregate over 100ms timescales)

early negative membership confirmation. The performance results of the IXP implementation are presented in Table 3.

Number of Hash Levels	All-Miss Cache Throughput
0	990 Mb/s
1	868 Mb/s
2	729 Mb/s
3	679 Mb/s
4	652 Mb/s
5	498 Mb/s

Table 3: Performance of Bloom Filter cache in worst case possible configuration

The IXP is far from an ideal architecture to implement a Bloom filter, in large part due to its lack of small, high-speed bit-addressable on-chip memory. Since there is no memory cache, data must be retained in on-chip registers during processing. The small number of available registers limits the performance of more complex tasks, which can be seen by the sharp drop-off in performance of a 5-level Bloom filter. Ideally, a Bloom filter would be implemented in hardware that supports parallel access on bit-addressable memory [29]. A simple and cheap custom ASIC can be constructed to implement a Bloom filter effectively.

This implementation uses the hardware hash unit. In this case, one hash is as difficult to calculate as four, because we simply use different portions of the generated hash to implement multiple hash functions.

Next generation IXP2000 hardware will feature 2560 bytes of on-chip memory per micro-engine, with up to 16 micro-engines per board. The memory access time will be 3 cycles, a vast improvement over the 16-20 cycles latencies of IXP1200 SRAM. A 2560 byte Bloom filter can store 467 elements. The next-generation micro-engines have a concept of "neighbouring" so that micro-engines can easily and quickly pass packet-processing execution to the next micro-engine "in line". This could allow for a high-speed implementation of a

Bloom filter where each micro-engine performs one or two memory look-ups so that the costs of a Bloom filter could be distributed across all the micro-engines.

VII. Dealing with Misclassification

The immediate question that arises when we introduce the possibility of a misclassification is to account for the result of the misclassifications. Let us first consider the case for a firewall.

If $F_1, F_2, \ldots, F_q$ unique flows $(q \leq L)$ were to set bits in the Bloom filter that matched the signature to a new flow, F', we will accept F' as a previously validated flow.

In the case that F' is a valid flow, no harm is done, even though F' would never have been analyzed by the packet classifier. If F' is a flow that would have been rejected by the classification engine then there may be more serious repercussions - the cache would have instructed the firewall to admit a bad flow into the network.

This case can be rectified for TCP based flows by forcing all TCP SYN packets through the classification engine.

Another solution would be to periodically reclassify packets that have previously been marked as cached. If a misclassification is detected, all bits corresponding to the signature of the flow id could be zeroed. This approach has the drawback of initially admitting bad packets into the network, as well as causing flows which share similar flow signatures to be reclassified.

If an attacker wanted to craft an attack on the firewall to allow a malicious flow, F'', into the network, they could theoretically construct flows, $F_1, F_2, \ldots, F_L$, that would match the flow signature of F''. If the firewall's internal hash functions were well known, this could effectively open a hole in the firewall.

To prevent this possibility, constants internal hash functions should not only be openly advertised, just as it is inadvisable to publish private keys. An additional measure would be to randomly choose the hash functions that the firewall uses. New hash functions can easily be changed as the Bloom filter ages.

In the case of a router, a misclassified flow could mean that a flow is potentially misrouted, resulting in an artificially terminated connection. In a practical sense, the problem can be corrected by an application or user controlled retry. In the case of UDP and TCP, a new ephemeral port would be chosen, and network connectivity can continue.

If we randomly force cached flows to be re-classified, we can reduce this "fatal" error to a transient one. TCP retransmit, and application-level UDP error handlers may make this failure transparent to the user.

The severity of these errors must be taken in into the context of the current Internet and TCP. To prevent IP spoofing attacks, TCP uses a 16-bit randomized initial sequence number. An attacker can already guess an initial sequence number of a TCP stream with a success rate of 1 in 2^{16}.

UDP packets are not even required to maintain a data checksum. In the Linux implementation of the network stack, even corrupt UDP packets are passed to the application.

VIII. Future Work

The aging functions discussed in this paper are inefficient, in the sense that it under-utilizes the Bloom filter's memory address. In the case of the cold cache algorithm, the Bloom filter is emptied. The Bloom filter is in a constant state of being under-utilized. Using the double-buffered algorithm introduces redundancy through the duplication of data. As in the cold cache algorithm, the Bloom filters are also under-utilized. It is possible that an algorithm based on randomly zeroing bits may prove to be an effective aging function – in this manner, we may be able to take advantage of the knowledge that a Bloom filter at optimum performance has ½ of its bits set.

Many of the implementation details of our architecture share common characteristics with IP traceback. Designing a system to support traceback in addition to caching could prove successful.

Using an approximate caching system presents us with a unique opportunity to dynamically balance the trade-off between accuracy and performance. In more traditional caches, during times of high load, cache performance decreases due to increasing cache misses. Intuitively, this behaviour may be sub-optimal. Ideally, performance should increase with workload. Our usage of a Bloom filter presents us with the opportunity to increase the effective size of the cache without consuming more memory by simply increasing the misclassification probability. This allows us the opportunity to increase cache performance in response to high amounts of traffic. Although more packets are misclassified, even more packets would be correctly forwarded. This may be better than the alternative – dropping more packets in response to increasing work.

The goal of the feedback system should be to balance the misclassification probability, p, with an acceptable cache performance/hit rate, h. To quantify this balance, we construct a "desirability" function, $f : (p, h) \rightarrow [0,1]$, where $f(1,0) = 1$ and $f(0,1) = 1$. The shape of function f must be chosen by the network administrator, to reflect the operator's preference in balancing hit rate and misclassification rate. The function f should be a monotonically increasing function for a constant p, and monotonically decreasing for a constant h.

Thus, we can view the choice of p as the result of a feedback system. A feedback controller would monitor the performance of the cache, and tune p, with the explicit goal of maximizing f.

IX. Conclusion

Typical packet classification caches trade-off size and performance. In this paper, we have explored the benefits that introducing inaccuracy has on packet classification caches. Using a modified Bloom filter, we have shown that allowing a small amount of misclassification can decrease the size of packet classification cache by almost an order of magnitude over exact caches without reducing hit rates. With the

deployment of IPv6 and the storage required to support caching of its headers, such a trade-off will become increasingly important.

ACKNOWLEDGMENTS

We would like to thank Brian Huffman and Ashvin Goel for helping us out with the analysis, and Damien Berger for helpful suggestions in programming the IXP. We would also like to thank Buck Krasic for his guidance, Jim Snow for his insight and Mike Shea, Jie Huang, Brian Code, Andrew Black, Dave Meier, Sun Murthy, William Howe, Sven Berger and our anonymous reviewers for their comments.

REFERENCES

[1] Bloom, B. H. Space/time tradeoffs in hash coding with allowable errors. Communications of ACM 13, 7 (July 1970), 422-426

[2] Baboescu, F. and Varghese, G., Scalable Packet Classification. In Proceedings of ACM SIGCOMM 2001, pages 199-210, August 2001.

[3] Broder, A., and Mitzenmacher, M. Network applications of Bloom filters: a survey. 40th Annual Allerton Conference on Communication, Control, and Computing, Allerton, IL, October, 2002

[4] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed Content Delivery Across Adaptive Overly Networks. In Proceedings of ACM SIGCOMM 2002, pages 47-60, August 2002.

[5] Carter, L., and Wegman, M. Universal classes of hash functions. Journal of Computer and System Sciences (1979), 143-154.

[6] Chiueh, T. and Pradhan, P. High Performance IP Routing Table Lookup using CPU Caching In Proc. of IEEE INFOCOMM'99, New York, NY, March 1999

[7] Chiueh, T. and Pradhan, P. Cache Memory Design for Network Processors, Sixth International Symposium on High-Performance Computer Architecture (HPCA 2000)

[8] claffy, k. Internet Traffic Characterization, Ph.D. thesis, University of California, San Diego, 1994

[9] Czerwinski, S., Zhao, B. Y., Hodes, T., Joseph, A. D., and Katz, R. An architecture for a secure service discovery service. In Proceedings of MobiCom-99, pages 24-35, N.Y., August 1999

[10] Estan, C. and Varghese, G. New directions in traffic measurement and accounting. In Proceedings of the ACM SIGCOMM 2002, pages 323-336, October 2002.

[11] Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. Longest Prefix Matching using Bloom Filters, In Proceedings of the ACM SIGCOMM 2003, pages 201-212, August 2003.

[12] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. ACM Trans. on Networking 8, 3 (2000), 281-293.

[13] Feldmann, A., and S. Muthukrishnan, Tradeoffs for Packet Classification, IEEE INFOCOM, 2000

[14] Feng, W., Kandlur, D., Saha D., Shin, K., Blue: A new class of active queue management algorithms, U. Michigan CSE-TR-387-99, April 1999

[15] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995

[16] Fraleigh, C., Moon, S., Diot, C., Lyles, B., and Tobagi, F. Packet-Level Traffic Measurements from a Tier-1 IP Backbone. Sprint ATL Technical Report TR01-ATL-110101, November 2001, Burlingame, CA

[17] Gopalan, K. and Chiueh, T. Improving Route Lookup Performance Using Network Processor Cache. In Proceedings of the IEEE/ACM SC2002 Conference

[18] Gupta, P., and McKeown, N. Algorithms for packet classification, IEEE Network Special Issue, March/April 2001, vol. 15, no. 2, pp 24-32

[19] Huitima, C. IPv6: The New Internet Protocol (2nd Edition). Prentice Hall, 1998.

[20] Iannaccone, G., Diot, C., Graham, I., and N. McKeown. Monitoring Very High Speed Links. In Proceedings of ACM SIGCOMM Internet Measurement Workshop, San Francisco, CA, November 2001

[21] Intel IXP1200 Network Processor, http://www.intel.com/design/network/products/npfamily/ixp1200.htm

[22] Jain, R., Characteristics of destination address locality in computer networks: a comparison of caching schemes, Computer Networks and ISDN Systems, 18(4), pp. 243-254, May 1990

[23] Lakshman, T. V., and Stiliadis, D., High-speed policy-based packet forwarding using efficient multi-dimensional range matching, In Proceedings of the ACM SIGCOMM 1998, pages 203-214, August, 1998

[24] McCreary, S., and claffy, k. Trends in wide area IP traffic patterns a view from Ames Internet exchange. In ITC Specialist Seminar, Monterey, California, May 2000

[25] Mitzenmacher, M. Compressed bloom filters. In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (2001), pp. 144-150.

[26] Partridge, C., Carvey, P., et al. A 50 GB/s IP Router. IEEE/ACM Transactions on Networking

[27] Partridge, C. Locality and route caches. NSF Workshop on Internet Statistics Measurement and Analysis (http://www.caida.org/outreach/isma/9602/positions/partridge.html), 1996.

[28] Passive Measurement and Analysis Project, National Laboratory for Applied Network Research (NLANR), available at http://pma.nlanr.net/Traces/Traces/

[29] Sanchez, L., W. Milliken, A., Snoeren, F. Tchakountio, C. Jones, S. Kent, C. Partridge, and W. Strayer. Hardware support for a hash-based IP traceback. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition, June 2001.

[30] Snoeren, A. C., Partridge, C., Sanchez, L. A., Jones, C. E., Tchakountio F., Kent, S.T., and W. T. Strayer. Hash-based IP traceback. In Proceedings of the ACM SIGCOMM 2001 Conference, volume 31:4 of Computer Communication review, pages 3-14, August 2001

[31] Srinivasan, V., Varghese, G., Suri, S. and Waldvogel, M. "Fast and Scalable Layer Four Switching" Proceedings of ACM SIGCOMM 1998, pages 191-202, September, 1998

[32] Stone, J., Partridge, C. When the CRC and TCP checksum disagree, In Proceedings of the ACM SIGCOMM 2000 Conference (SIGCOMM-00), pages 309-319, August 2000

[33] Squid Web Proxy Cache, http://www.squid-cache.org

[34] Qiu, L., Varghese, G., Suri, S. Fast firewall implementations for software and hardware-based routers. In Proceedings of ACM SIGMETRICS 2001, Cambridge, Mass, USA, June 2001.

[35] Xu, J., Singhal, M., and Degroat, J. A novel cache architecture to support layer-four packet classification at memory access speeds, In Proceeding of INFOCOM 2000, pp. 1445-1454, March 2000.