Good Memories: Enhancing Memory Performance for Precise Flow Tracking

Boris Grot, William Mangione-Smith
University of California, Los Angeles
Department of Electrical Engineering
{bgrot, billms}@ucla.edu

Abstract

Flow tracking is an integral aspect of many network-related applications, including intrusion detection, network administration, per-flow billing, and network engineering. While some applications do not require precise tracking of every packet or flow and successfully employ statistical sampling or approximations to achieve an acceptable level of accuracy, many others, especially in the security domain, do not have such flexibility. Faced with rapidly growing number of networked devices, increasing Internet usage, and rising network bandwidth, precise flow tracking has become a very difficult problem with system memory as the main bottleneck.

In this paper, we look at memory performance when tracking a large number of flows, show that finding an existing flow in memory often requires multiple key comparisons, and examine ways to reduce the number of comparisons for existing flows and avoid them altogether for the majority of new flows. Additionally, we propose a novel scheme called Predictive Placement that reduces the number of searches requiring three or more comparisons from 4.87% with a linked list implementation to just 0.0027% of all lookups. These results are based on our simulations with widely used long packet traces from CAIDA. Our approach relies on a Bloom filter derivative that encodes each element's location using multiple bitmaps. The accuracy of the scheme depends on the number of bitmaps employed and on their load factor.

1. Introduction

Traffic measurement and monitoring is necessary for managing today's high-speed links. Tracking flows is one crucial aspect of such management, and has applications in network security, administration, engineering, and accounting (in the form of usage-based billing) [1]. A number of recent works have investigated approximate and statistical approaches to flow tracking at high link speeds, focusing on counting the number of flows [2], measuring flow volumes [6][8], identifying the "elephants" [1] and "superspreaders" [17], and classifying flows [7]. Unfortunately, while many network-related tasks can tolerate small errors and approximations, others can not.

Network intrusion detection is one such application which requires precise flow tracking, as certain attacks (i.e., stealthy port scans, TCP connection hijacking, evasion by fragmentation), can *only* be detected by keeping exact and complete per-flow state [3]. In fact, most, if not all, of today's Network Intrusion Detection Systems (NIDS) keep per-connection state [4].

The current crop of NIDS aims to protect enterprise and campus-wide networks from attacks. For instance, product literature for NetScreen-IDP 1000 [18], a top-of-the-line intrusion detection and prevention system from Juniper Networks, specifies maximum throughput of 1 GB/s and a maximum number of supported sessions of 500,000 with 4 GB of memory[1]. However, our tests using traces of real traffic showed periods of over 700,000 flows simultaneously live with data rates well below 1Gb/s, indicating that NetScreen might be overwhelmed by the number of connections, and an attack might go unnoticed. Another study [15] reported a day-time average (*not* peak) of over 450,000 flows on a very modest 155 Mb/s (OC-3) link. Thus, keeping track of a sufficient number of flows is a challenge even at moderate data rates.

This challenge will likely be exacerbated going forward, as NIDS are pushed deeper into the network, where they are exposed to more flows (allowing them to better identify, and react to, attacks) and higher link speeds [3]. In order for these systems to maintain real-

[1] It is not known whether the limitation in the number of simultaneous sessions is due to system scalability issues or memory capacity

time performance, flow identification and matching to tracked flows will have to scale with the growing demands. In fact, McKeown, et al, identify this classification as "the most expensive and complex step" of a flow monitoring system, requiring "a one-to-one mapping between a fixed set of packet fields (the 5-tuple made of protocol number and source and destination addresses and port numbers) and the memory portion that contains the flow record" [16].

The contribution of this work is in investigating ways to reduce the number of comparisons that a packet entering the system must undergo before being mapped to a correct flow. After an overview of our experimental setup, we examine the conventional approach to tracking flows, consider the performance impact of inserting new flows at the head of their respective hash bucket versus appending them at the tail, evaluate the use of a Bloom filter for avoiding memory accesses for new flows, and present a novel scheme for matching packets to existing flows so as to significantly reduce the number of key comparisons that must be made.

2. Methodology

We conducted our study using three anonymized traces from an OC-48 link [19]. Each trace file is around 9 GB and represents a 60-minute snapshot of the network. Two of the traces were collected in August of 2002, while the third one was collected in January 2003. Table 1 lists some of the details for each trace.

Since the goal of the study is to enable applications that rely on precise flow tracking to meet the demands of high-bandwidth networks, we chose to support a reasonably large number of flows. Assuming 512 MB of system memory and a 64-byte flow descriptor (large enough to store the 5-tuple key, time stamp, and some application-specific data), we arrived at 8 million flows. This leaves plenty of room for scaling by adding more memory, as the effective bandwidth in our traces never exceeds 500 Mb/s.

One of the first steps in looking up a flow is producing a hash of its key (see Section 3); hence, an efficient hashing function is crucial. Such a function must be easy to implement in hardware, be sufficiently fast in operation, and approach a uniform random mapping. The ability to change the function over the lifetime of the hardware would also be very desirable, as it would make the system less vulnerable to exploits targeting specific memory locations or access patterns.

The function we chose has all of the above properties and is defined [10] as

$$H_q(x) = (x_1 * q_1) \wedge (x_2 * q_2) \wedge \ldots \wedge (x_i * q_i),$$

where '*' denotes bit-wise AND while '^' denotes bit-wise XOR operations. Here, x_i is the i^{th} bit of the key, and q_i is a bit string stored in register *i*. Thus, to obtain a 32-bit hash value from a 64-bit key, we would need 64 registers, each 32 bits wide. It is worth pointing out that the hash function itself is not fixed in hardware, as changing it requires simply initializing the register array to a different value.

Flow aging is another issue that has drastic implications on system performance. A flow that has not seen any packets after a certain time should be evicted from memory to make space for new flows and reduce the length of the search for incoming packets. A long timeout value may unnecessarily fill the memory with a large number of dead flows. A short timeout, on the other hand, may result in premature evictions of long-lived flows, leading to inaccuracies in the collected data or, in intrusion detection applications, to undetected attacks. Several studies [15][16] have noted that a timeout value of around 60 seconds offers a good balance between accuracy and memory requirements, leading us to adopt it for our experiments. In practice, the exact timeout value should be determined based on each application's needs and the underlying network's characteristics.

Note that if a newly arrived packet maps to an expired flow that has not yet been evicted, the flow will get "revived" as its last-accessed time stamp will get updated. On the other hand, if the flow has been deleted, a new flow will get allocated for the packet. In this case, the flow will effectively be counted twice. This effect is reflected in Table 1, which lists the number of flow allocation events over the duration of each trace, rather than a count of unique flows in the file.

In our current simulation environment, the memory scrub task is modeled as a method call that takes zero simulation time; however, in a real system such a process could be implemented as a low-priority background task, gradually walking over the memory space when the memory and the bus are otherwise idle.

An important concern that must be addressed is whether the memory subsystem will have sufficient bandwidth to handle the expected traffic. For instance, the latest DDR2 DRAM chips are able to support 533 MT/s (millions of transfers per second) at 64 bits per transfer. Intel's 915G chipset paired with this type of memory achieves single-channel peak bandwidth of 4.25 GB/s [22]. Assuming a 1 Gb/s link carrying a constant stream of 64 byte packets (worst case scenario which maximizes the memory workload) and 64 byte

Table 1: Trace characteristics

Trace	Name	Number of packets	Number of flows	Avg number of live flows	Max number of live flows
1	20020814-090000-0-anon.pcap	272,551,909	16,194,780	427,528	559,971
2	20030115-100000-1-anon.pcap	236,556,354	14,613,711	405,478	714,166
3	20020814-110000-0-anon.pcap	314,235,548	15,502,808	409,062	550,950

flow descriptors, the memory configuration described above has enough bandwidth to handle over 30 64-byte accesses per packet. The number drops to just 3.4 accesses on a 10 Gb/s link; however, memory throughput can be scaled by adding additional channels. Thus, a quad-channel configuration could support 17 GB/s peak transfer rate, enough for over 13 64-byte transactions for every minimum-sized packet received.

A recent study of memory controller performance in server environment [23] found that an out-of-order controller subjected to a representative server workload was able to achieve sustained throughput of over 5 GB/s with a DDR2 memory subsystem rated at 6.4 GB/s peak bandwidth, which is only 22% lower than the theoretical peak rate. This leads us to conclude that a memory subsystem with sufficient bandwidth for flow tracking at high data rates is, indeed, practical.

3. Evaluation

The common approach to tracking flows is to compute a hash of several fields from the packet header and use the resulting value to index a memory holding the full key and the desired data. Since multiple flows are likely to produce the same hash value, a linked list of descriptors is kept in memory. Such an approach is used by the FreeBSD TCP/IP stack, which maintains a hash table with each element (hash node) pointing to a linked list [20]. The hash key is usually taken to be a 5-tuple composed of the source and destination IP addresses, source and destination port numbers, and the protocol field.

As discussed in Section 2, we simulated a memory with a maximum capacity of 8 million flows. Since one of the best ways to reduce the number of hash collisions is to increase the size of the hash space, we logically partitioned the memory into 512 K hash buckets with a maximum list length of 16 elements Although the maximum list length does not have to be limited, potentially leading to some very long lists, this is highly undesirable as the worst-case search length becomes intolerable. The latter is simply disastrous in a latency-sensitive environment such as a core- or edge-based network element. On the other hand, fixing the list length to a very small value will likely lead to collisions and dropped flows, as new flows will compete with existing flows for list space..

Thus, in case a new flow must be added to a list which has reached its maximum length, our approach is to first try to locate an expired entry in the list and remove it; if no element in the list has timed out, the element closest to timing out is evicted instead.

3.1 List Insertion Position

The first set of experiments we conducted compared insertion of a new flow at the head of the linked list versus at the tail (traversals always start at the head). The choice is not obvious. On one hand, it is well known that the majority of the flows are short-lived and carry only a small fraction of the total traffic on a link. For instance, one study found that nearly 80% of all flows are active less than one second, after which they go completely silent [14]. This characteristic would suggest that inserting new flows at the tail would be a good strategy, as they are likely to be short-lived and should not increase the access latency for long-lived flows. On the other hand, network traffic exhibits fairly good temporal locality; specifically, packet trains, defined as two or more back-to-back packets belonging to the same flow, have been observed for a number of server workloads [20], suggesting that inserting at the head of the linked list would better exploit this temporal locality.

Figure 1 plots the number of key comparisons for the three traces using head and tail insertion. Since each comparison requires reading the flow descriptor from memory, the number of comparisons is effectively the number of independent memory accesses. The trend is clear: insertion at the head produces fewer comparisons over the duration of the traces than insertion at the tail. The savings produced by inserting at the head of the list are 14.89%, 12.83%, and 9.89% for Traces 1, 2 and 3, respectively, with an

average of 12.42% fewer comparisons. Thus, we do not further consider end-of-list insertion and adopt a linked list-based implementation with head-of-list insertion as our baseline.

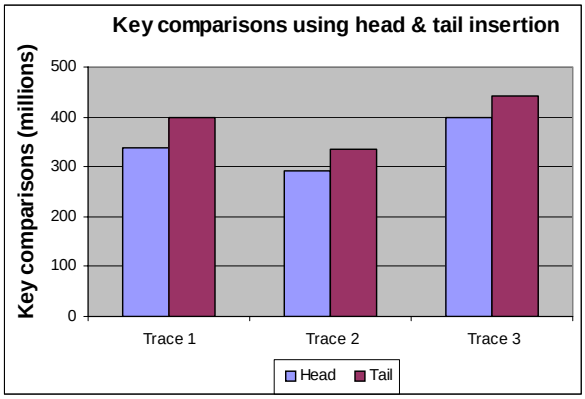


Figure 1: Head vs. tail insertion: number of key comparisons

3.2. Distribution of Search Lengths

Figure 2 plots the number of searches of a given length using head-of-list insertion – note that the y-axis is logarithmic.

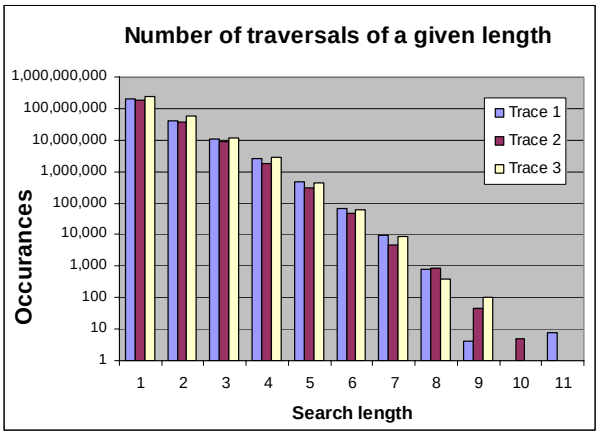


Figure 2: Number of linked list traversals of a given length

As expected, the greatest majority of all lookups require only one or two comparisons. However, over 40 million packets in the three traces combined – nearly 5% of the total packet count – result in three comparisons or more, with a few requiring up to 11 comparisons. Recall that our memory is partitioned into 512 K buckets, so given that we rarely see over 500 K flows simultaneously live in our system, such long comparison sequences might seem surprising at first. But with several hundred million packets processed per trace, and given occasional spikes in traffic, these events do in fact occur. As such, the rest of the study focuses not just on reducing the number of key comparisons required to match a packet to a flow, but specifically on eliminating these long search sequences.

3.3. Avoiding List Traversal for New Flows

One obvious source of maximum-length list traversals are new flows. Consider what happens when a packet from a new flow arrives: as usual, memory is indexed using the hash of the packet's key, the linked list pointed to by the source node is reviewed from beginning to end, and only after failing to find a matching flow is a new flow descriptor allocated.

In our traces, the ratio of packets without a matching flow to the total number of packets processed is fairly small – only 6% of packets require a new flow to be allocated. However, since each new flow is accompanied by a complete traversal of the associated list, we wanted to study the impact that new flows have on the number of key comparisons.

3.3.1. Identifying New Flows Using Oracle Knowledge

Table 2 shows the number of key comparisons that could be avoided using oracle knowledge of whether a given flow is in memory. Note that the actual reduction in the number of comparisons is somewhat small – around 3.3% on average. This is explained by the fact that we are using a fairly large and sparsely populated memory – exactly what one would like to have to reduce the number of hash collisions. In fact, most of the time, the number of tracked flows is just over 400,000 in a memory with capacity for 8 million flow descriptors, with the implication that a new flow often maps to an empty list. Hence, in many cases, no key comparisons need to be made when a new flow is inserted.

The intuitive conclusion supported by our simulations with a smaller – and higher utilized – memory is that greater memory utilization will significantly increase the number of comparisons caused by unmatched flows. So despite the fact that potential savings in the number of memory accesses are quite modest under the presented setup, the fact that each new flow is a longest sequence lookup with

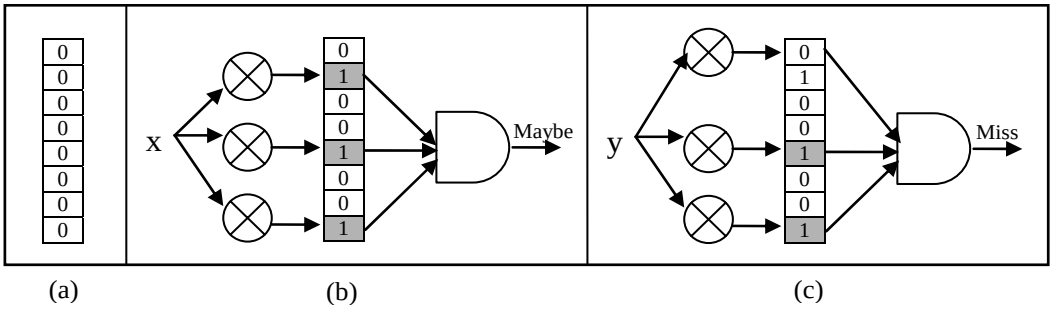


Figure 3: An empty Bloom filter (a); a hit in the Bloom filter (b); a definite miss (c)

respect to its associated list length motivated us to try to reduce the impact of such worst-case offenders.

Table 2: Reduction in key comparisons using oracle knowledge of whether a flow is in memory

Trace	Actual key comparisons	Avoidable using oracle knowledge	Longest Search Length
1	338,517,750	12,335,783	11
2	291,025,359	10,522,455	10
3	399,490,102	11,328,868	9

3.3.2. Identifying New Flows Using A Bloom Filter

Bloom Filter Theory

A Bloom filter [5] uses an array of bits to represent a set of elements. Each member is represented by k bits, where each bit is indexed by a separate hash function. If one or more of the k bits corresponding to an element is '0', the element is guaranteed not to be a member of the set. If, on the other hand, all k bits are set, the element belongs to the set with the probability of $1 - f$, where f is the false positive ratio. In effect, a Bloom filter is a compact representation of a set, allowing queries for membership that sometimes yield a false positive, but never a false negative.

Figure 3 shows an example of a small Bloom filter that represents each element using three bits ($k = 3$). In (a), an empty Bloom filter is depicted. Figure 2 (b) shows a hit in the Bloom filter for some key X, after X has been inserted into the filter. X is hashed using three different hash functions, whose outputs are used to index the Bloom filter. Since the values at all three accessed locations are '1', the filter produces a hit. Note that a hit in the Bloom filter is not a definite 'yes' but is more akin to a 'maybe'. This is similar to conventional hashing, where two hash values that match might have been computed from the same key, but a comparison of the keys is required to confirm a match. Part (c) shows a miss in the Bloom filter. Even though two of the three bits are set, there is no chance that Y is an element of the given set.

Now, the relationship between the false positive rate, f, and the number of bits used for each element, k, is expressed by the following formula:

$$f = (1 - e^{-nk/m})^k \qquad (1)$$

where m is the size of the Bloom filter (in bits) and n is the number of member elements. Thus, the product $n*k$ is the number of bits occupied by the n members of the Bloom filter. It can be shown that f is minimized with respect to k when

$$k = (m/n) \ln 2 \qquad (2)$$

which corresponds to the false positive ratio

$$f = (1/2)^k \qquad (3)$$

In fact, comparing this result to (1), we see that the false positive ratio is minimized with respect to k when approximately half of the bits are set. Since in a network setting we have no control over how many elements are in the set (i.e., how many flows we are tracking), the filter should be sized so that it gives an acceptable false positive ratio in the worst possible case.

Our Implementation

In our setting, the worst case represents tracking 8 million flows, so the Bloom filter should have around 16 million bits for a reasonable false positive rate when the memory is at full capacity. Since the Bloom filter is supposed to identify new flows to avoid the long-latency search through the entire list, its accuracy is most important when the memory is highly utilized and lists are long.

Table 3: Bloom filter performance

Trace #	False positive %	% of new flows correctly identified	# of key comparisons eliminated	Realized % of oracle's potential
1	0.0019%	99.96%	12,328,818	99.95%
2	0.0022%	99.97%	10,517,355	99.95%
3	0.0016%	99.97%	11,323,073	99.95%

A value for f around 10-20% seems to be appropriate, given that in our traces only 6% of all packets do not match an existing flow. This means that in practice, the Bloom filter should produce a false positive for just 0.6-1.2% of all packets when tracking the maximum number of flows, since only new flows can result in a false positive. Of course, a more lightly loaded Bloom filter would be expected to have an even higher accuracy. Thus, we use three hash functions ($k = 3$) with the theoretical f of 12.5%

One problem with a conventional Bloom filter is that it is impossible to remove an element from the set, since doing so requires clearing all the bits associated with that element. Because bits can be shared among multiple set members (it is the combination of all k bits which uniquely identifies a member), clearing all bits associated with an element is not an option. For this reason, we employ a counting Bloom filter [9], which is implemented with a vector of counters, rather than bits. Thus, adding an element requires incrementing all k counters, while evicting it results in the counters being decremented.

With the Bloom filter in place, each arriving packet generates the following sequence of events: the 5-tuple key is hashed using three different hash functions, whose outputs serve as indices into the Bloom filter array. The three counters at the indexed locations are read out and their values are compared to '0'. If all counters are greater than '0', then the given flow is likely found in main memory and a search through the appropriate linked list is initiated. However, if one or more counters are '0', the packet is immediately tagged as a new flow, its corresponding counters in the Bloom filter are incremented, and a flow descriptor is inserted at the head of the appropriate linked list.

For faster access, the Bloom filter array can be implemented in SRAM. Assuming 8 bit counters, a Bloom filter with 16 million entries would require 128 Mb. This is somewhat large for a single SRAM part; fortunately, the Bloom filter array can be divided into disjoint sub-arrays, so that each hash function addresses its own physical memory. An additional benefit of this approach is that all memories can be accessed in parallel.

Bloom Filter Performance Evaluation

Table 3 summarizes our finding. As expected, our Bloom filter has a very high accuracy rate, as evidenced by the percentage of new flows correctly identified. Equally important is the fact that nearly all the savings offered by an oracle predictor are being realized, eliminating virtually all key comparisons for new flows. The false positive rate is extremely low due to three factors: appropriate Bloom filter sizing, its low occupancy ratio, and relative infrequency of occurrence of new flows.

Unfortunately, despite its excellent ability to correctly identify new flows and eliminate the associated memory accesses, the Bloom filter offers little help in terms of reducing the total number of long list traversals. Figure 4 shows the effect of the Bloom filter on the number of key comparisons. The two spikes in Trace 1 correspond to a reduction in searches of length 9 from four occurrences to one, and the elimination of the sole occurrence of a 10-long search. However, for the three traces, most searches of length greater than 1 see a very modest reduction of around 4% in the number of occurrences.

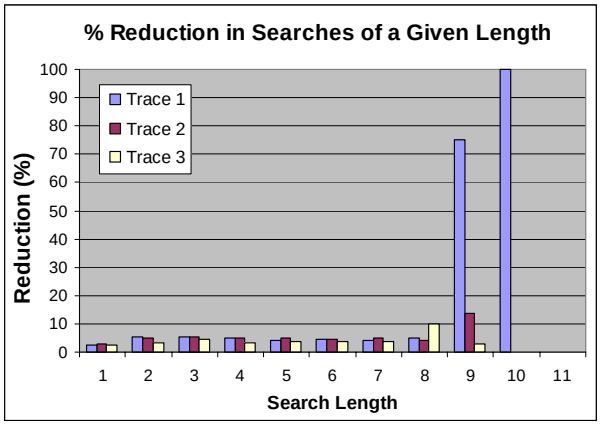
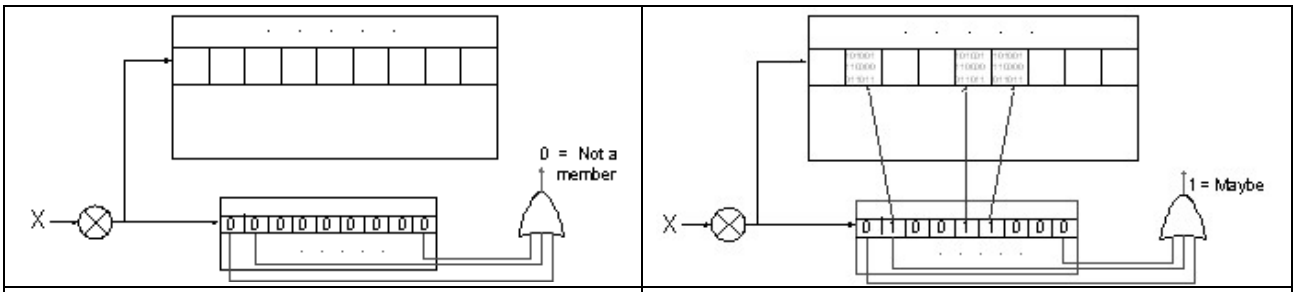


Figure 4: Percent reduction in list traversals of a given length



(a) (b)

Figure 5: Replacing a linked list with an array. (a) shows an empty hash bucket; since none of the bits are set, X is definitely not an element of the set. (b) shows a non-empty hash bucket with three potential candidates at offsets 1, 4, and 5

3.4. Predictive Placement

The biggest shortcoming of the Bloom filter is its limited utility. While it helps eliminate lookups for new flows that enter the system, all subsequent packets belonging to a flow must go through the conventional lookup process. Furthermore, the memory requirements of a Bloom filter are relatively high, especially in light of its limited usefulness. Additionally, the ability to "blindly" insert new flow descriptors has a price, as linked list lengths must be explicitly tracked to ensure that a given list has not reached its maximum length and a new flow may be added.

Ideally, we would like a way to not only predict whether a given flow is already being tracked, but also to locate it in memory without doing all, or most, comparisons associated with a linked list traversal. A good solution should have a reasonable memory footprint with the ability to trade accuracy for memory overhead.

Predictive Placement Overview

Consider a memory partitioning scheme which allocates a fixed number of contiguous locations to each hash bucket, in effect forming a two-dimensional array. Hence, each hash value is effectively a pointer to a one-dimensional array in memory. Furthermore, consider a bitmap with the same number of bits as there are array elements, logically organized as the memory array and indexed with the same hash function. Thus, each entry in the array has a corresponding bit set in the bitmap. A simple way to determine if a given element X is in memory is to compute a hash of its key and check the bits in its respective hash bucket in the bitmap. If none of the bits are set, X is definitely not in memory. Alternatively, if some of the bits are set, then each bit represents a potential memory location where X may reside. In the former case, the bitmap obviates the need for a memory access, while in the latter it provides a list, in the form of a bit vector, of possible locations for X. If the bit vector has non-zero entries, then verifying X's existence requires checking the memory array at indices corresponding to these entries. Figure 5 (a) and (b) depicts the two cases just described.

So what, if any, benefits are there to this organization compared to a conventional linked list approach? In reality, the two are quite similar, as the list of potential locations for X produced by the bitmap is hardly different from a linked list which must be traversed one entry at a time. One potential benefit of the bitmap approach is that fetches of multiple candidate locations can be overlapped, something that's normally not possible with a conventional linked list. This is because obtaining a pointer to a given element in a linked list first requires fetching its predecessor, whereas a bitmap can be used to determine the exact offsets at which memory must be read. However, we will not take this advantage into consideration in this study, although the potential benefit is not insignificant and is something we are currently exploring.

Now, instead of using one bitmap as described above, consider using two. The goal is to eliminate at least some of the bits in the candidate bit vector as potential locations of X. As before, the bitmaps are logically organized like the main memory, with each hash bucket mapping to a fixed number of contiguous bits. One bitmap is still indexed with the same hash function as the memory, providing a one-to-one mapping between non-zero bits and allocated memory entries. We call this bitmap the *primary filter*. The second bitmap, which we will refer to as the *secondary filter*, is indexed by a completely different hash function.

Under the new organization, X might reside in memory if and only if both bitmaps contain a '1' in the same position. Thus, a membership query consists of a bitwise AND operation on the respective hash buckets in the primary and secondary filters, followed by an OR of the result vector. If the OR operation produces a '1', a potential hit, which we call a *maybe*, is registered and memory locations corresponding to non-zero entries in the result vector are fetched. Figure 6 shows an example of a membership query.

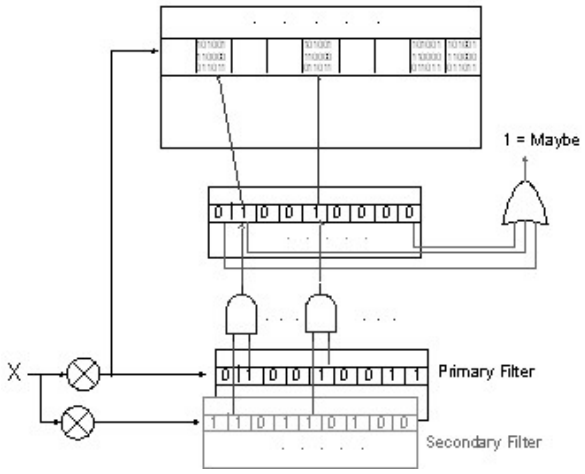


Figure 6: Replacing a linked list with an array and multiple bit maps. A potential hit ('Maybe') occurs when both primary and secondary arays have 1's in the same position.

Insertion of a new element into the memory is similarly easy and requires finding bits with a value of '0' in the same position in both bitmaps. An insert operation sets both bits to '1', while a remove operation clears the bits. Note that in the case of insert, the selected memory location is guaranteed to be empty due to a one-to-one mapping between the primary filter and the memory.

Additional Considerations

One issue that must be considered in using Predictive Placement is that of a bucket in either the primary or the secondary filter becoming full. In the former case, a full bucket indicates that a given memory array is at its maximum capacity. As discussed in Section 3, dealing with this situation requires walking over the entries in the bucket and replacing either an expired entry or another element via some LRU-like approach.

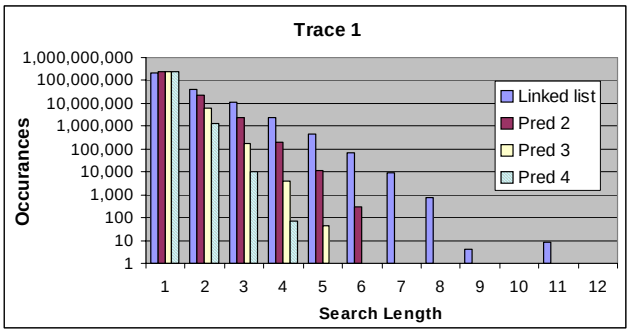
The case of a secondary filter filling up is a little trickier. Since inserting an element requires setting a bit and removing it results in the bit getting cleared, sharing bits among multiple elements is not an option. Of course, the situation is identical to that of a conventional Bloom filter. The solution there was to use counters instead of bits, and that's what we chose to do here. Thus, each hash bucket in the secondary filters is actually a vector of counters. We get excellent results using just two bits per counter. It is important to note that counters are only required for the secondary filters, since the primary filter is a one-to-one mapping with memory.

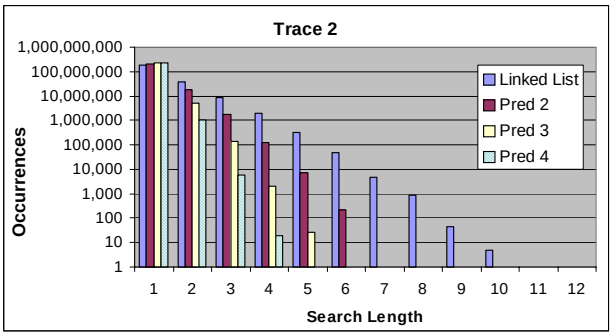
With this minor modification, let's consider the memory requirements of Predictive Placement. As before, memory is organized into 512 K hash buckets with a fixed number of contiguous memory locations per bucket. Since our lists are limited to 16 entries, each hash bucket has exactly this number of available slots. Primary and secondary filters mirror the memory organization; with 512 K 16-bit buckets, the primary filter requires only 8 Mb of storage. Due to the use of counters, secondary filters require 2 bits per entry, so their memory footprint is 16 Mb. Thus, a primary filter with two secondary filters needs only 40 Mb of storage – quite reasonable given today's SRAM technology. Even more pleasant is the fact that each filter naturally lands itself to being in a separate physical device; in fact, increasing accuracy simply requires adding another secondary filter SRAM.

Another issue that must be discussed is the worst-case performance of our approach. The extreme case is that of the memory being completely full, with every bit in the primary filter set and every counter in every secondary filter greater than 0. Clearly, in this situation, the filters will return a *maybe* answer for every query. Hence, worst-case performance of Predictive Placement is comparable to that of a linked list approach, since every query will require an in-order traversal over the appropriate hash bucket. Indeed, like any other hashing-based scheme, including the Bloom filters, our approach requires data structures to be under-utilized for satisfactory performance. And like a Bloom filter, our design offers a trade-off between space requirements and accuracy.

Evaluation

We tested our approach using a primary filter with 1, 2, and 3 secondary filters, where each secondary filter is indexed with a different hash function. Generally, we anticipate that accuracy will improve with a higher number of secondary filters as aliasing between different elements is reduced.





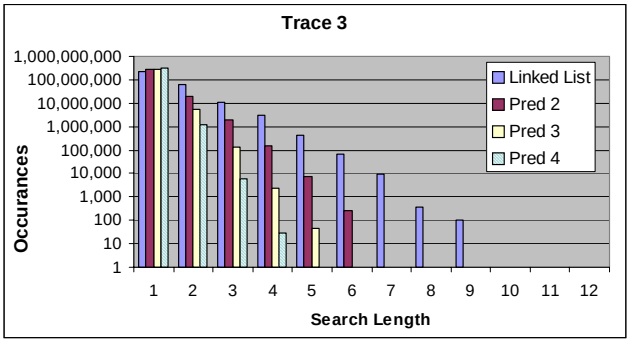


Figure 7: Number of list traversals of a given length for a Linked List implementation and Predictive Placement with 2, 3, and 4 filters (primary + 1 or more secondary filters) for Trace 1, Trace 2, and Trace 3

Figure 7 compares the number of searches of a given length for a linked list using head-of-list insertion against our Predictive Placement approach. As expected, each additional filter used in the Predictive Placement scheme provides a noticeable improvement in prediction accuracy. Pred 2, the simplest implementation with just one secondary filter, eliminates all searches of length 7 and above and provides an order of magnitude reduction in the number of searches of length 3 and above. In fact, the number of occurrences of these searches is cut from over 40 million in a linked list implementation (4.87% of all lookups) to 6.7 million (0.8%) when using Pred 2. Adding an additional secondary filter (Pred 3) results in the elimination of length 6 traversals and drops the number of searches requiring 3 comparisons or more to approximately 461 thousand (0.06%).

Pred 4, which uses three secondary filters, not only completely eliminates all searches of length 5 and above, but reduces the number of length 4 traversals from well over a million to under 100 for all three traces. The total number of searches of length 3 or more is just 22 thousand or 0.0027% of all lookups.

Table 4 provides a detailed breakdown of the number of searches of each length for Trace 3 using a linked list and Predictive Placement schemes. Notice that with Predictive Placement, the number of traversals of length greater than 1 drops, while the number of traversals of length equal to 1 increases. This, of course, is the desired outcome, as it indicates that long searches are replaced by lookups that match on the first comparison.

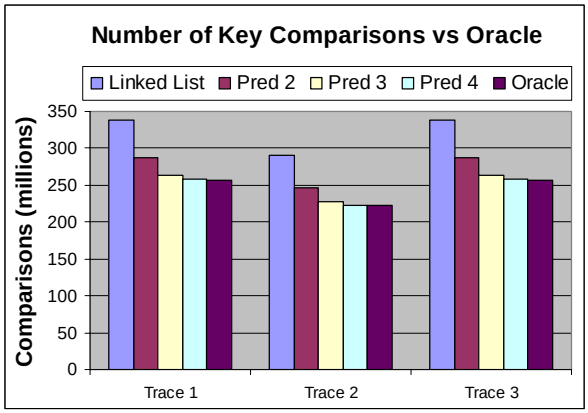


Figure 8: Number of key comparisons for Traces 1, 2, and 3 using a Linked List and Predictive Placement with 2, 3, and 4 filters against an Oracle predictor

We also compared the performance of our approach to an oracle predictor. Unlike the oracle introduced is Section 3.3, this predictor not only recognizes new

Table 4: Number of searches of a given length for Trace 3

	1	2	3	4	5	6	7	8	9
Linked List	232,766,358	59,009,640	11,482,698	2,904,605	437,253	64,288	8,860	374	105
Pred 2	278,819,287	20,403,004	1,991,816	146,140	7,373	248	0	0	0
Pred 3	293,674,215	5,538,158	137,026	2,473	44	0	0	0	0
Pred 4	297,687,903	1,156,562	5,758	30	0	0	0	0	0

flows and avoids the associated lookups, but always finds an existing flow on the first try. Results are presented in Figure 8. A separate calculation shows that Pred 4 is within 0.6% of the total number of key comparisons required by the oracle predictor for all three traces.

4. Related Work

A number of recent works have employed Bloom filters for network-related tasks. Several of these used Bloom filters or similar data structures for measuring flow volumes [1][6][8][21]. These approaches do not require precise flow tracking, have limited utility, and are generally not 100% accurate due to the false positives produced by the Bloom filter.

Chang, Feng, and Li utilized a Bloom filter to assist in packet classification [7]. Classified packets were added to the Bloom filter to avoid future classification, while unclassified packets went through the regular classification process. This is similar to our use of a Bloom filter to avoid linked list traversals for new flows. Their work also explored encoding information into a Bloom filter for the purpose of packet routing. However, unlike flow tracking, routing requires a static mapping between a given flow and an output port. Our work uses a modified Bloom filter structure to efficiently find an empty memory location to which a new flow may be written and supply a list of candidate locations in which an existing flow may be found.

One possible alternative to linked lists is closed hashing. Open hashing, sometimes referred to as chaining, resolves collisions by keeping a linked list pointed to by an entry in the hash table. This is the technique used in our study. Closed hashing, on other hand, resolves hash collisions by probing additional locations in the table, often through the use of a separate collision resolution function. Double hashing is an example of this approach, and its use in flow monitoring applications has been suggested by McKeown et al [16]. While closed hashing has good theoretical average-case performance for lightly-loaded hash tables, it is unsuitable in an environment with frequent deletions, since these tend to increase the average search length [11]. Closed hashing also does not permit optimizations such as head-of-list insertion.

Another well-known technique for improving lookup performance is caching. Most cache-related studies in the networking domain have focused on improving the performance of route lookup and packet classification, with reported cache hit rates ranging from 60% to over 90% [12][13]. In fact, we consider our work orthogonal to caching for several reasons. In network intrusion detection, for instance, caching cannot be relied on to satisfy any fraction of memory requests, since it makes the system susceptible to attacks against the cache with the goal of degrading memory subsystem performance.

Additionally, caching in the traditional sense implies a fixed tag and data arrangement, where the tag is a portion of the memory address at which the data resides. However, in a conventional flow tracking application, given a newly arrived packet, the memory address of its corresponding flow is typically not known. This implies that the key used to identify a flow should itself serve as the tag. But different applications might require different definitions of a flow, leading to different key size requirements. For example, an application collecting per-flow statistics on a network node might use a five-tuple (104 bits) for flow identification, whereas an algorithm for identifying distributed port scans may very well use destination IP addresses (32 bits) as the flow id. Naturally, data size requirements for these applications would also be expected to vary widely, leading to difficulties implementing a conventional cache in an adaptable multi-purpose architecture. Our approach helps eliminate long probes to memory regardless of whether a cache is employed in the system, enabling a wide range of latency- and bandwidth-sensitive applications requiring flow tracking.

5. Conclusion

In this paper, we evaluated techniques for reducing the number of key comparisons required to find a matching flows in applications requiring precise flow tracking. We established that head-of-list insertion reduces the number of key comparisons required to find a matching flow by an average of 12.4% for the three traces when compared to insertion at the tail of a linked list. Using a memory 16 times larger than the average number of live flows in the system and head-of-list insertion, we witnessed frequent occurrences of searches involving more than one key comparison. Almost 5% of all packets required three key comparisons or more, with maximum observed search lengths exceeding eight comparisons in all three traces.

We evaluated the use of a Bloom filter to eliminate lookups for new flows entering the system. The reduction in the number of key comparisons turned out to be a modest 3.3%, with most searches of length greater than 1 seeing a 4% decrease in the number of occurrences.

We then introduced our Predictive Placement approach, which uses multiple filters to reduce the number of candidate locations that must be searched. The filters use a Bloom-like data structure to encode not only membership but also the location of every element. Using a primary filter and one secondary filter, we succeeded in removing all searches requiring more than six comparisons and reduced the number of searches of length 3 and above to approximately 0.8% of all lookups. A Predictive Placement design with three secondary filters completely eliminated searches with over four comparisons and reduced the number of probes of length 3 or more to just 0.0027%. In fact, the number of key comparisons required by this scheme was within 0.6% of an oracle predictor which always finds an existing flow in exactly one lookup and avoids lookups altogether for new flows.

6. References

[1] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. *Proc. of the ACM SIGCOMM 2002*, pp. 323-336, October 2002.

[2] C. Estan, G. Varghese and M. Fisk. Bitmap algorithms for counting active flows on high speed links**.** *Proc. of the 2003 ACM SIGCOMM conference on Internet measurement (IMC-03)*, pp. 153-166, ACM Press, October 2003.

[3] K. Levchenko, R. Paturi, and G. Varghese. On the Difficulty of Scalably Detecting Network Attacks. *Proc. of the Eleventh ACM Conference on Computer and Communication Security*, October 2004.

[4] R. R. Kompella, S. Singh and G. Varghese. On Scalable Attack Detection in the Network. *Proc of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 187-200, 2004.

[5] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), pp. 422-426, July 1970.

[6] A. Kumar, J. Xu, L. Li and J. Wang. Space-code bloom filter for efficient traffic flow measurement. *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pp. 167-172, 2003.

[7] F Chang, W. Feng, and K. Li. Approximate Caches for Packet Classification. *Proc. IEEE INFOCOM 2004*, March 2004.

[8] S. Cohen and Y. Matias. Spectral bloom filters**.** *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pp. 241-252, ACM Press, 2003.

[9] L. Fan, J. Almeida and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Network*, 8(3), pp. 281-293, 2000.

[10] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, pp. 1378-1381, December 1997

[11] Weiss, M.A., *Data structures and algorithm analysis in C++*. 1994.

[12] P. Newman, G. Minshall, and L. Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, January 1997.

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

[14] N. Brownlee. Some Observations of Internet Stream Lifetimes. *Passive and Active Measurement Workshop*, 2005.

[15] J. Apsidorf, K.C. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, Affordable, High Performance Statistics Collection. *Proc. of the 10th USENIX conference on System administration*, pp. 97-112, September 1996.

[16] G. Iannaccone, C. Diot, I. Graham and N. McKeown. Monitoring very high speed links. *ACM Sigcomm Internet Measurement Workshop*, November 2001.

[17] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. *Network and Distributed System Security Symposium*, February 2005

[18] Juniper Networks NetScreen-IDP 10/100/500/1000. http://www.juniper.net/products/intrusion/dsheet/110010.pdf

[19] CAIDA. Anonymized OC48 traces. http://www.caida.org/data/.

[20] L. Zhao, S. Makineni, R. Illikkal, R. Iyer, L. Bhuyan. Efficient Caching Techniques for Server Network Acceleration. *Advanced Networking and Communications Hardware Workshop (ANCHOR 2004)*, June 2004.

[21] D. Nguyen, J. Zambreno and G. Memik. Flow Monitoring in High-Speed Networks using Two Dimensional Hash Tables. *Proc. of Field-Programmable Logic and its Applications (FPL)*, Aug.-Sep. 2004.

[22] Intel 915G/915GV/910GL Express Chipset Memory Configuration Guide. *Intel White Paper*. September 2004.

[23] C. Natarajan, B. Christenson, F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-Processor Server Environment. *Proc. of the 3rd workshop on Memory Performance Issues: in Conjunction with the 31st International Symposium on Computer Architecture*, 2004.