Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines

Flavio Bonomi
Cisco Systems, Inc.
flavio@cisco.com

Michael Mitzenmacher*
Harvard University
michaelm@eecs.harvard.edu

Rina Panigrahy†
Stanford University
rinap@cs.stanford.edu

Sushil Singh
Cisco Systems, Inc.
sushilks@cisco.edu

George Varghese
Cisco Systems, Inc./UCSD
varghese@cs.ucsd.edu

ABSTRACT

Many networking applications require fast state lookups in a concurrent state machine, which tracks the state of a large number of flows simultaneously. We consider the question of how to compactly represent such concurrent state machines. To achieve compactness, we consider data structures for Approximate Concurrent State Machines (ACSMs) that can return false positives, false negatives, or a "don't know" response. We describe three techniques based on Bloom filters and hashing, and evaluate them using both theoretical analysis and simulation. Our analysis leads us to an extremely efficient hashing-based scheme with several parameters that can be chosen to trade off space, computation, and the impact of errors. Our hashing approach also yields a simple alternative structure with the same functionality as a counting Bloom filter that uses much less space.

We show how ACSMs can be used for video congestion control. Using an ACSM, a router can implement sophisticated Active Queue Management (AQM) techniques for video traffic (without the need for standards changes to mark packets or change video formats), with a factor of four reduction in memory compared to full-state schemes and with very little error. We also show that ACSMs show promise for real-time detection of P2P traffic.

Categories and Subject Descriptors: C.2.6 Internetworking : Routers

General Terms: Algorithms, Measurement, Design.

Keywords: Bloom filters, state machines, network flows.

1. INTRODUCTION

In this paper, we introduce the idea of an Approximate Concurrent State Machine (ACSM), which aims to track the simultaneous state of a large number of agents within a state machine. Concurrent state machines arise naturally in many networking applications, especially in routers, where one wishes to track the behavior of many simultaneous flows.

Our motivation for considering approximate versions of concurrent state machines arises in two ways. First, we observe that in the last few years, routers and networking devices have increasingly begun to keep state about TCP connections. One powerful motivation for this was the advent of packaged firewalls and Intrusion Detection devices that keep state for each TCP connection in order to detect security violations. This was followed by application level load balancers, and then application level QoS devices that attempt to look at application headers in order to provide more discriminating QoS to applications. While the security and application level QoS applications are well entrenched in the market, there have been more recent proposals to do video congestion control [9] and for identifying Peer-to-Peer (P2P) traffic [10], both of which can also be loosely placed in the Application QoS category.

While Application Level QoS can theoretically be provided by marking packets appropriately, such marking requires standards changes. Further, much more sophisticated QoS state machines can be implemented by keeping state for each application flow. Thus, in practice, many networking devices keep state for each TCP connection and the trend shows no sign of abating. If the state kept for each connection is small (say less than 20 bits) compared to the TCP 5-tuple (roughly 100 bits), then it is natural to speculate whether one can reduce the overall state required by eliminating or reducing the space required for the flow identifier.

Reducing state is crucial to high-speed routers because it allows state machines to be implemented on-chip without resorting to slow off-chip memories. For example, for a router keeping track of 1 million connections (a number found in many studies such as [23]), using 100 bits per connection requires 100 Mbits of memory, which is impractical using on-chip memory (or any form of SRAM). However, if the state were dialed down by a factor of 5 to 20 Mbits, this becomes at least technologically feasible.

Reducing memory by removing or reducing the identifier size has an obvious analogy with Bloom filters [2]. Thus, our second motivation for considering ACSMs arises from the re-

[*] Supported in part by NSF grant CCR-0121154 and a research grant from Cisco.

[†] Part of this work was done while working at Cisco Systems.

SIGCOMM'06, September 11–15, 2006, Pisa, Italy.
Copyright 2006 ACM 1-59593-308-5/06/0009 ...$5.00.

markable success of Bloom filters in networking. From 1970, when Burton Bloom first described a compact probabilistic data structure that he used to represent words in a dictionary [2], until 1995, there was little interest in using Bloom filters for networking. However, judging from the number of papers that make direct or indirect use them [4], interest in such applications has surged in the last 10 years.

A Bloom filter is essentially a compact representation of a set. Standard exact representations of sets such as hash tables and binary trees require at least L bits per element to be stored, where L is the size of element, and often require additional space for pointers. By contrast, a Bloom filter is an inexact representation of a set that allows for "false positives" when queried (in other words, it can sometimes say that an element is in the set when it is not). In return, it allows very compact storage: roughly 10 bits per element for a 1% false positive probability, independent of the size of the elements in the set or the size of the set itself.

Given that memory appears plentiful today, it may seem surprising that Bloom filters have become so popular. However, Bloom filters allow economical solutions to various kinds of scarcity, including:

- Bandwidth: There are several situations where network bandwidth is still expensive. For example, distributed caching is useful when bandwidth is limited, but in that case sending lists of all the cached items in messages can also be expensive. The Summary Cache paper [8] uses Bloom filters to compactly represent URLs stored at a cache in inter-cache messages.

- High Speed Memory: While ordinarily DRAM memory is cheap, fast SRAM memory and especially on-chip SRAM continues to be comparatively scarce. For example, using on-chip Bloom filters can greatly speedup a naïve string matching scheme, requiring access to slower off-chip memory only rarely [7].

- Memory for Very Large Sets: There are some sets whose sizes are astronomical, so that they cannot be stored even when considering the cheap price of DRAM today. For example, Snoeren et al attempt to solve the problem of packet traceback after an attack by storing a complete log of all packets received in the last hour at every router. The only way to make this even barely technologically feasible is to represent the logs by a Bloom filter [21].

While Bloom filters provide a technique for *set membership* lookups, the vast majority of lookups in networking applications, whether in routers or endnodes, are actually *state* lookups. While route lookups and TCP state lookups cannot tolerate the loss of fidelity caused by false positives, we have already argued that there are several emerging uses, many in the loose area of Application Level QoS, that can benefit from state reduction and can tolerate the loss in fidelity. Further, guided by the analogy with Bloom filters, it is plausible that there will be other applications (besides the ones suggested in this paper) that will be found by other researchers.

Thus we seek a compact structure like a Bloom filter, but for the setting of state lookups. In this setting, Bloom filters themselves have many limitations (besides false positives), including:

- **No associated value:** A Bloom filter determines whether an element is in a set; it does not return state associated with an element. Chazelle et al. have generalized Bloom filters to functions that can return a (small) set of values [5], but this techniques does not allow state changes (see Section 2).

- **Deletion is expensive:** A Bloom filter allows easy insertion but not deletion. Deletions in a Bloom filter are handled using a counting Bloom filter, which keeps a counter instead of a single bit at each hash location [8, 15]. Counters are incremented on an insertion and decremented on a deletion. Unfortunately, using counters increases the size of the filter by a nontrivial factor; if too few bits are used in the counters overflows and false negatives can arise.

- **No notion of time:** A key property of network state machines (e.g., the TCP state machine) is that they allow the state to be timed-out. This is often essential as the only way to deal with failure in networks and distributed systems is to infer failure by the lack of message arrival within a specified timeout period. Bloom filters have no notion of time or timeouts, and a naïve association of every filter element or cell with a timer would greatly increase the space required.

In this paper, we present new techniques to design Approximate Concurrent State Machines that address each of the limitations described above. We start by considering very simple techniques based on Bloom filters, as we believe this would be the natural starting point for people considering this problem. Perhaps surprisingly, our comparison of several techniques suggest that in most cases, an approach based on fingerprints and *d-left hashing* [3, 24] performs best and offers the most flexibility. Our techniques still have false positives, with a small probability; under some circumstances, they may also yield false negatives, or return the wrong state. We further introduce another type of error condition, corresponding to a "don't know" response, which is generally less damaging than other types of errors. Thus our ACSMs are suitable only for applications that can tolerate a small probability of error. Despite this weakness, we suggest that ACSMs can enable more stateful (but still lightweight) processing at routers.

One key feature of our structures is that we turn a disadvantage of this setting into an advantage. Handling deletions is often expensive, because of the need for counters to do deletion properly. But in state-lookup situations, there is generally a natural time-out, where the corresponding flow should be deleted. We can utilize the time-out mechanism for flows that time out to also delete flows that terminate successfully at a later time, removing the need for counters, and saving space. We also utilize the fact that many networking applications can tolerate some lack of precision in the timer value, especially if the timer is used to reclaim inactive state. Providing precise timers (e.g., for retransmission) will require extensions to our data structure.

To demonstrate the power of ACSMs, we study a specific, novel example application: discriminate dropping for video congestion control. Using ACSMs, a router can efficiently keep track of the current frame status of MPEG video packets, allowing for more discriminating drop policies during times of congestion. We describe experiments to show the effectiveness of such control compared to the naïve dropping schemes that are the only recourse of existing routers, and show that the implementation costs are low. We also briefly describe an experiment to suggest the promise of ACSMs for real-time detection of P2P traffic [10]. More generally, we introduce ACSMs as a useful tool for other applications that can tolerate a small probability of error in return for a compact representation.

While the two example applications we study can roughly be described as techniques to provide Application Level QoS (where some errors can be tolerated), an important application we do not study in this paper is for analysis of network traffic. Network monitors are often used to answer questions about network traffic; using compact ACSMs may allow faster response to complicated queries using state machines with only a small loss in accuracy. Further, even when doing queries using disk logs of network traffic (e.g., NetFlow records), ACSMs may help in 2-pass algorithms that first sift out candidate flow records in the first pass (including false positives) using ACSMs that fit in memory, and then weed out the small number of false positives in a second pass. We believe that ACSMs can play an important role in measurement infrastructure in software and hardware.

We also note that a variation of our fingerprint/hashing scheme provides an alternative approach for creating a counting Bloom filter [8, 15] that uses much less space (a factor of 2 or more) than the standard construction. Counting Bloom filters have many potential uses, but tend to be expensive with regard to space; our alternative may therefore also be useful for many applications.

To summarize, our contributions are:

- The introduction of the ACSM problem.
- The introduction (and comparison) of several solutions to the ACSM problem starting with a simple solution that utilizes Bloom filters, followed by a solution that extends Bloom-filter like techniques, and ending with a solution that uses a combination of hashing and fingerprint compression, and is *very* different in spirit from the techniques underlying Bloom filters.
- A novel construction for counting Bloom filters.
- Techniques for taking advantage of time-outs for space savings.
- Experimental evaluation of ACSMs for multiple problems, including an application for congestion control on MPEG streams.
- The potential use of ACSMs for speeding up measurement algorithms working on large data sets.

2. RELATED WORK

We are aware of previous papers that extend Bloom filter techniques to realize *particular* state machines; for example, there are papers that consider structures for approximately storing multisets, keeping an approximate counter for each element. In terms of *general* state machines, the most relevant previous work is the *Bloomier filter* [5]. While a Bloom filter is designed to represent a set, a Bloomier filter is designed to represent a function on a set. Specifically, we have a set of items S, and associated with each item $x \in S$ is a value $v(x)$ that we assume is in the range $[1, V]$. For each item $x \notin S$, we take $v(x) = \text{null}$. (Here null should be distinct from the number 0, which we may use for other purposes.)

A Bloomier filter always returns $v(x)$ for $x \in S$; when $x \notin S$, null should be returned with high probability. Just as a Bloom filter can have false positives, a Bloomier filter can return a non-null value for an element not in the set. Previous work on Bloomier filters has focused on the setting where the set S is static. When both S and the corresponding function values are static, there are linear space solutions that seem viable, although they have not been tested in practice. There are lower bounds that show that when S is static and function values are dynamic, non-linear space is required [17].

Another closely related work is [13], which takes a first step towards state machines by allowing classes (which can be thought of as the state of an element). While [13] also describes ideas for efficient deletion, the paper does not combine deletion and classes as would be required to support state machines. Further, the technique works by coding across individual Bloom filters; in order to reduce false negatives, the resulting codes tend to increase the number of parallel lookups required (some of the codes described in [13] require 1000-fold parallelism) which may be hard to achieve in practice. The idea of returning a Don't Know condition also appears in this paper.

Other alternative constructions with improved functionality over Bloom filters and Bloomier filters have recently been suggested [17]; however, while the theory of these alternatives has been studied, at this point we are not aware of any practical implementations or uses. These constructions are much more complicated than the simple but effective hashing methods underlying Bloom filters. We consider this simplicity a virtue for practical use, and aim to keep our structures similarly simple when possible.

3. THE STATE MACHINE SETTING

In order to provide as concrete an analysis as possible, we specify the setting for our analysis and experiments. We are given a single state machine, and a collection of *flows*. We work in a streaming model, where our data stream consists of a sequence of state updates for the collection of flows. A flow has an associated *flow-id* to identify the flow. A flow becomes extant when initiated, which we model as beginning at some initial state, and ceases to be extant when it is terminated, which we model as reaching some terminal state. Each extant flow is associated with a current state. The data stream provides *transitions*, corresponding to (flow-id, string) pairs; that is, there is a function from state-string pairs to states corresponding to the transitions of the state machine. For example, a transition for a flow could take the form "Go to state 7,", or "If in state 5, go to state 3," or "Add 1 to the current state." We desire a data structure that will track the state for each flow efficiently, reacting to each transition in the stream.

Initially, we will consider systems that are *well-behaved*, in the following sense: every flow is properly initiated by an initiation transition, every transition requested for every flow is valid, and every flow is (eventually) properly terminated by a termination transition. In many real-world situations, systems are not entirely well-behaved. When we consider *ill-behaved* systems, we must consider that an adversary or errors within the system may lead to faulty behaviors, including specifically: transitions may be introduced into the stream for non-extant flows; a flow transition might not be valid (in which case we assume the flow should remain in its current state); and a flow might not terminate properly. For example, in a network measurement application, when the analysis starts it may have to deal with flows that were already active before the measurement started. We note that well-behaved systems are much easier to analyze formally.

In this setting, there are four natural operations that we desire of our data structure:

- *InsertEntry(flow, state)*
- *ModifyEntry(flow, newstate)*
- *Lookup(flow) outputs (state)*
- *DeleteEntry(flow)*

When faced with a possible state transition, one can perform a lookup to find the current state for a flow, determine the subsequent state according to the transition, and modify the state accordingly. (Alternatively, the current state of the flow could be given as part of the input to the Lookup operation; we discuss this further below.)

There are various types of errors that can occur. Suppose a lookup is done on a non-extant flow, i.e. a flow that has not been properly initialized, and the result is that it appears to have a valid state. We call this a *false positive*. Note that this should be distinguished from a false positive on a Bloom filter (that may be a part of an ACSM); this is a false positive on a flow, and we will strive to keep the terminology clear. Suppose that no state is given for an extant flow. We call this a *false negative*. If an erroneous state is given for an extant flow, we call this a *false return state*. Finally, we introduce a new type of error that we think is interesting and useful in its own right; in particular, it allows us to avoid the lower bounds determined by work on Bloomier filters. We allow that for an extant or non-extant flow the data structure returns a“don’t know” (hereon, abbreviated as DK) state. Returning DK is generally far less damaging for an application than a false negative or a false return state. For example, it may be that the concurrent state machine is used as a fast path for packet classification, but in the case of a DK response a slower path classification routine can be used. Obviously, we want all types of errors to occur with very small probability, including DK errors. The best tradeoff among the different types of errors is highly application dependent, suggesting that data structures that allow such tradeoffs are more valuable.

As a concrete example, consider using an ACSM for identifying Peer-to-Peer (P2P) Traffic as we do in Section 5.2.2 in order to rate-limit such traffic. A false-positive implies that we will (wrongly) rate-limit the traffic that is not P2P. While this is clearly bad, our approach uses existing heuristics [11] that already have false positives. Similarly, false negatives imply that we miss some P2P traffic, but then so does the existing approach. In this application, if no action is taken on a DK, a DK can at most increase the false negative probability. The bottom line is ACSMs may be justified if the space for the application can be reduced considerably at the cost of a very small increase in the false negative and positive rate. This is especially so if the reduction in space allows the application to be done in faster memory.

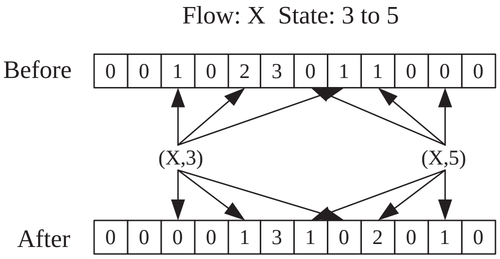


Figure 1: A state change with a DBF ACSM (assuming the old and new states are given). The counters for the old flow-state pair are decremented, the counters for the new flow-state pair are incremented.

3.1 A Direct Bloom Filter Approach

There is a simple approach directly using a Bloom filter to obtain a concurrent state machine, which we call the direct Bloom filter (DBF) ACSM. The current set to be stored consists of (flow-id, current state) pairs. We assume the state is represented as a value in the range $[1, V]$. Because we will want to handle deletions, we describe the approach using counting Bloom filters; alternatively, deletions may also be accomplished using timing-based mechanisms, as we describe in Section 3.2 below. This approach, while seemingly obvious and natural, is quite limited and recommended only in very special cases.

Let us first consider the situation when the system is well-behaved. Insertion is straightforward. Lookup operations are easily done if one is also given a state as input; that is, one can check if a flow is in a specific state easily by checking for the appropriate (flow-id,state) pair. However, if the state is not part of the lookup input, then one has to check all possible states. In this case, the time to check for a state is proportional to the number of states; moreover, because of false positives in the Bloom filter, it is possible that a flow appears to be in multiple states, in which case a DK must be returned. (This situation would also be problematic when having to delete a flow; either the state must be given, or if the flow appears to be in multiple states, a deletion cannot be accomplished without risk of error, so timing-based methods must be used.) This approach is therefore most suitable when either there is a very small number of states, or the transitions themselves specify the start state and end state for the transition. Many state machines have the property that the current state is naturally encoded as part of the transition. For example, when state transitions are unique, so that the input that causes the state to change from A to B is unique to both of the states A and B, then this information informs the lookup. Deletions of flows are also straightforward using a counting Bloom filter when the state is also given. Modifying an entry corresponds to a lookup, deletion, and insertion of a new (flow-id,state) pair. An example of modifying an entry is given in Figure 1.

Analysis of the DBF ACSM is straightforward, assuming that states are given as part of the input when performing a deletion. If there are n extant flows , with m counters and k hash functions used in the filter, the probability of a false positive f is just that for a standard Bloom filter [4, 16]:

$$f \approx \left(1 - (1 - 1/m)^{kn}\right)^k \approx (1 - e^{kn/m})^k.$$

It is well known that for a fixed value of m and n, the optimal choice for k to minimize f is $(m/n)\ln 2$, giving $f \approx (0.6)^{m/n}$. For a lookup on an extant flow with no state information as part of the input, the probability of returning a DK with s states is the probability of not having a false positive for the flow on the $s-1$ other states, or (when $f << 1/(s-1)$)

$$1 - (1 - f)^{s-1} \approx f(s-1) \approx s(0.6)^{m/n}.$$

We now consider problems that may arise in a system that is not well-behaved. A false positive for a non-extant flow can occur, if the underlying Bloom filter gives a false positive. Notice that a false positive can have ongoing effect in the following case: if a false positive causes an invalid state transition to occur, it will change the filter. The changes may later cause future false negatives, false positives, false return states, or DK results; such dynamic interactions are naturally hard to analyze systematically. A flow-level false negative can also occur for an extant flow if an invalid state transition is attempted, and a false positive from the Bloom filter makes it appear that the flow is in the wrong state. (This cannot happen if all possible states are tested, as then a DK should be returned, but if the transition specifies the initial state, this may not be done to save time, as previously explained.) Non-terminating flows eventually would cause the filter itself to saturate with non-zero entries, unless timing-based mechanisms are used.

3.2 Timing-Based Deletion

Before describing improved structures, it seems best to describe our approach to timing-based deletion in some detail in the context of DBF ACSMs. They will work in entirely similar ways for our other proposed structures.

We have seen that non-terminating or otherwise ill-behaving flows require a mechanism for cleaning out of the data structure after a certain amount of time. Similarly, in cases where deletions might not normally be possible (such as if the state is not given for a deletion request, and a flow appears to be in multiple states according to the filter), a timing-based deletion mechanism will ensure that an uncompleted deletion eventually happens.

A straightforward method uses a single flag bit per cell and a single global counter, and breaks time into phases, where a new phase begins for example after a certain number of operations. The counter tracks the number of operations until a new phase begins. At the beginning of a phase, all flag bits should be set to 0. During a phase, every cell that is touched has its flag bit set to 1. The appropriate definition of touched may depend on the context; in our settings, it is best to say that a cell is touched if it is used in a lookup or insert operation, or if the cell value is modified, but not if the cell is deleted. At the end of a phase, all unflagged cells are reset to the appropriate value for a cell that has no flows hashed to it; generally this is 0. Also, all flagged cells retain their value, and all flags are reset to 0. In this way, any cell not touched during a phase is reset, so that any flow that has not taken part in an operation will eventually be removed from the ACSM.

When using this timing-based approach, counters may no longer be necessary, greatly reducing the space required. For example, we described the DBF (direct Bloom filter) ACSM as using counters to allow deletions, but once we introduce this timing mechanism, there is no reason to use counters at all. The space might be better used by simply using a standard Bloom filter (with more bits). Alternatively, with the DBF ACSM, you might use a combination of counters and the timing scheme. A small counter would be useful when flows change state often within a timing phase; otherwise, the filter will be highly polluted. The timing-based deletion remove non-terminating flows. An example of a timer reset is given in Figure 2.

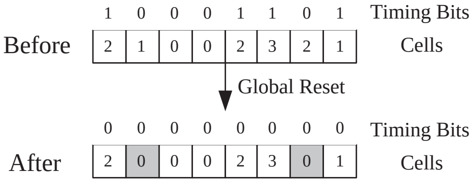


Figure 2: Example of timer-based deletion. One bit is used for each cell; when the global timer goes off, untouched cells (in grey) are reset to 0.

The aging process is hard to analyze without some necessarily inaccurate model of deletions; in this paper, we focus on studying the use of timers via experimental evaluation. We note, however, that the number of phases required before a cell resets itself to empty is essentially geometrically distributed. That is, if a flow is deleted, the probability that some new flow hits that cell in the next phase is easily determined given the number of new flows in that phase; assuming that the number of new flows is roughly the same from round to round, and that existing well-behaved flows tend to terminate within a round, the number of phases before a cell is untouched (and hence reset) is geometrically distributed. It follows that a well-behaved, sufficiently large filter will reach a fairly consistent steady-state over time.

3.3 A Stateful Bloom Filter Approach

The DBF ACSM is rather naïve; one might suspect that a similar structure making more careful use of the states would perform better. We now describe such an alternative structure, which we call a stateful Bloom filter (SBF) ACSM. Again the underlying structure is like a Bloom filter, but the Bloom filter cells are neither bits nor counters but instead a value corresponding to the state. This is similar in spirit to an idea used in recent hash table constructions [12], although the application is quite different.

Each of the m cells in our filter can store a value (in $[1, V] \cup \{\text{null}\}$) and a counter. Null with a 0 count represents no flows are currently hashed to the cell; we refer to this as 0 henceforth. Null with a count of 2 or more represents that two or more flows have collided at that cell and corresponds to a DK. Each flow is hashed to k cells (like a standard Bloom filter).

The main innovation of the SBF ACSM is that whenever we have a collision at a location in our filter among two or more flows with different state values, we encode "don't know" in the cell. When doing a lookup on a flow-id in the filter, as long as at least 1 value is not DK, a state value can be returned. (This threshold of 1 could conceivably be changed to trade off various errors; we have not found this

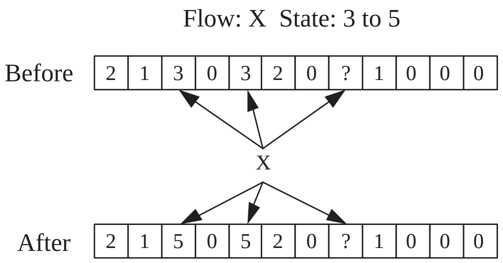


Figure 3: A state change with a SBF ACSM (no state needs to be given). The hash locations give the state or possibly a Don't Know (represented as a '?'). A cell obtains a '?' if on an insertion a state already is stored in the cell.

especially effective, and we do not consider it further here.)

We can define rules for the various operations. Below, where we say *hash the flow*, we mean the operation takes effect at each cell the flow hashes to.

- Insertion. Hash the flow. If the cell counter is 0, write the new value and set the count to 1. If the cell value is DK, increment the count. If the cell value equals the flow value, increment the count. If the cell value does not equal the flow value, increment the count but change the cell to DK.
- Modify. Hash the flow. If the cell value is DK, leave it. If the current count is 1, change the cell value. If current count is exceeds 1, change the cell value to DK.
- Deletion. Hash the flow. If the count is 1, reset cell to 0. If the count it at least 1, decrement count, leaving the value or DK as is.
- Lookup. Check all cells associated with a flow. If all cell values are DK, return DK. If all cell values have value i or DK (and at least one cell has value i), return i. If there is more than one value in the cells, the item is not in the set.

If the system is well-behaved, these rules guarantee that we never return an incorrect value for a flow, although we may return a "don't know". This structure could therefore be used as an alternative to Bloomier filter structures in a dynamic setting. Notice that once there is a collision in a cell, so that a DK value arises, one must wait for the cell count to go to 0 before the cell is reset from DK to 0. This requires that the system is well-behaved; otherwise, deletions must be handled via a timing-based mechanisms.

As with the DBF ACSM, we can provide an analysis for the SBF ACSM under the assumption the system is well-behaved. For convenience here we consider the case of insertions only; to handle deletions, one must have a model of deletions so as to account for the DK values as described above. Suppose we have m cells, n flows, and k hash functions. Further let n_i be the number of flows with state value i. The probability $x \in S$ with value i yields a don't know is equivalent to the probability that each of its cells are hashed to by some element with a value that is not i. This is then easily computed using the standard Bloom filter analysis as

$$\left(1-(1-1/m)^{k(n-n_i)}\right)^k \approx \left(1-e^{-k(n-n_i)}\right)^k.$$

The probability that $x \notin S$ yields a false positive with value DK is

$$\left(1-(1-1/m)^{kn}\right)^k-\left(\sum_i (1-1/m)^{k(n-n_i)}\left(1-(1-1/m)^{kn_i}\right)\right)^k,$$

as this is just the probability each cell for x is hashed to by some flow, subtracting off the probability each cell is hashed to only by flows with only one value. The probability that $x \notin S$ yields a false positive with value i can be approximated by assuming that x hashes to k distinct cells, finding the probability that each of the k cells gives either an i or a DK, and subtracting off the probability $x \notin S$ yields a false positive with value DK. (The expression is long but not complex.)

As before, when using timing, we do not need to keep counters, as every DK state is equivalent regardless of the count; we can simply wait for a timer event to reset a cell. In this case, we detect when two flows share a cell only on an insertion; on an insertion, any non-zero cell becomes a DK. Again, in this setting, an invalid state transition can incorrectly change cell values, leading to future errors. An example of a state change under a SBF ACSM (without counters) is given in Figure 3.

When the system is not well-behaved, there are further issues to deal with. As mentioned, if a flow is not properly terminated, then the filter will become polluted, causing increased DK return values. This is handled using timing-based mechanisms. Also, it is possible for spurious packets to disrupt the filter, by causing a state transition when one should not occur, although this can only happen if there is a false positive for a specific state value. The effects of this problem are less severe than for the direct Bloom filter approach; the most likely outcome is a false negative (rather than a false return state) as the cells for a flow may then not have matching states. Spurious packets can also introduce DK values similarly.

3.4 An Approach Using d-left Hashing

Although the SBF ACSM has reasonable performance, we have found that for most settings an approach using d-left hashing in combination with fingerprints gives better performance. We call this a fingerprint-compressed filter (FCF) ACSM. A great advantage of the FCF ACSM is that there are a few key parameters that can be fine-tuned for various performance tradeoffs. The application of d-left hashing in combination with fingerprints is interesting in its own right; for example, in Section 4 we also show how this technique can be used to obtain a data structure with the same functionality as a counting Bloom filter, using much less space.

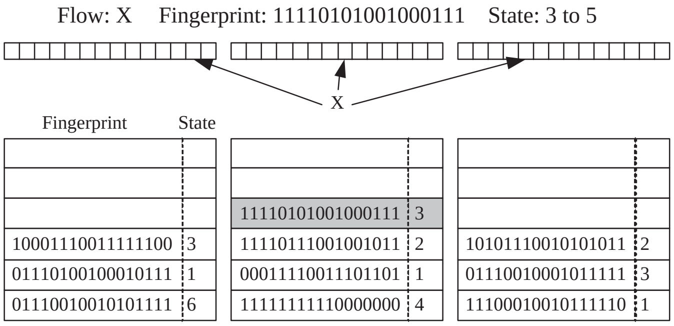
The basic idea is very simple: store a fingerprint of the flow-id along with the flow's current state in a hash table. If the set were static, and there was suitably efficient *perfect* hash function for the set of flows, this would suffice [3, 17]. (Recall a perfect hash function maps a fixed set to a range without collisions.) As we are in a dynamic setting, perfect hash functions are generally not efficient for the purposes we consider. We demonstrate that instead using d-left hashing (in combination with timing mechanisms) provides an efficient alternative. While the similarities between perfect hash functions and d-left hashing were noted previously in [3], this application appears entirely novel.

For good usage of space and quick, fixed lookup times, no pointers should be used. Instead, we adopt a hash table with a fixed size, and a fixed number of flows that can be stored in each hash bucket. We call this fixed number of flows the *height* of the bucket. Each bucket will therefore be assigned a fixed amount of space, corresponding to the number of flows that can be held. If fewer flows than the maximum are stored in a bucket, we assume that the remaining space is filled by empty flows, which are signaled in a specific way, say with an entry of all zeroes.

In order to efficiently use the space available in the hash table under these conditions, as well as make the probability of a bucket overflow appropriately small, we can apply d-left hashing, as explained in [3]. We provide a summary here. In the d-left scheme, the hash table is broken up into d subtables, ordered for convenience from left to right; generally these subtables are of equal size (although technically this is not necessary). When a flow is placed in the hash table, it has d possible locations, one in each subtable, with the location in each subtable given by independent uniform hashes of the item. (Many practical hash functions approximate this behavior reasonably in practice.) The d possible buckets are examined, and the item is placed in the bucket holding the fewest items; in case of ties, ties are broken to the left. The number of items in a bucket is also called its *load*. This is a particularly efficient variation of using multiple choices in hashing, giving extremely balanced loads, and in particular very small maximum loads.

Notice the number of different parameters and can choose in setting up an ACSM in this way: the number of hash functions d; the number of buckets b of each subblock of the hash table; the height h of each bucket; the size f of the fingerprint in bits. Assuming x additional bits for each flow (to represent the state and the timer bit) gives a total space of $dbh(f + x)$ bits for the hash table.

The settings of d, b and h must be such so that the probability of an overflow is very small. (We note that also a small TCAM could be added to handle hash table overflows if their probability is sufficiently small. While a small TCAM would be recommended in practice, appropriate design should make overflows extremely rare, as we describe, and we ignore this in the subsequent analysis.) The utilization u of the table is the fraction of occupied cells; if we have n flows and dbh cells then $u = n/(dbh)$. A typical configuration, given as an example in [3], considers the case where $d = 3$, $h = 6$, and $b = n/12$. This gives a utilization of $u = 2/3$. In this case, the asymptotic fraction of buckets that overflow (in the case of insertions only) is approximately $5.6 \cdot 10^{-31}$ [3]; even with insertions and deletions of items, overflow events are remarkably rare. This suggests that overflow, while it needs to be considered, can be handled straightforwardly with this structure. Also, higher utilizations and hence less overall space is possible by using larger values of d, f, and h (and correspondingly smaller values of b).



Fingerprint	State				
		11110101001000111	3		
10001110011111100	3	11110111001001011	2	10101110010101011	2
01110100100010111	1	00011110011101101	1	01110010001011111	3
01110010010101111	6	11111111110000000	4	11100010010111110	1

Figure 4: A state change with an FCF ACSM. First, the three buckets associated with flow X are found. When the appropriate fingerprint is found, the state can be changed. Alternatively, if the flow is deleted and re-inserted, it could be re-placed into the leftmost of the least loaded buckets.

Even in the case where flows are well-behaved, flows can yield false positives or DK values. A false positive occurs if a lookup is performed on a non-extant flow and a fingerprint matches. As each fingerprint matches with probability 2^{-f}, a simple union bound gives an upper bound of $dh2^{-f}$. Similarly, a DK value could arise if the fingerprint for a flow appeared in two buckets that the flow hashed to. Such an occurrence would also, of course, make it impossible to perform a deletion in a valid manner (as we would not know which entry to delete – this is why the problem is easier to handle with a static set and a perfect hash function). If a DK is found on a lookup, and a state change is required, it may be appropriate to change the state of both items to a special DK value; this depends on the application. DK values can also be handled using a TCAM, to store flow-ids that yield a DK in the data structure explicitly.

Finally, it is worth noting that on a state change, there are two possible ways of accomplishing the change. First, one could simply change the state of the appropriate fingerprint as found in the table. Second, one could delete the current state and re-insert the new state; this may cause a change in the location of the fingerprint, if there is an alternative bucket with a smaller load. The first approach requires less work, and the second approach will do a slightly better job keeping the load evenly distributed. See Figure 4 for an example.

As with our other schemes, issues become more complicated if the system is ill-behaved, as non-extant flows can change the state for extant flows. To cope with this, the fingerprint should be chosen to keep false positives sufficiently rare. Also, timing-based deletions must be used to remove the fingerprint of non-terminating flows.

4. AN ALTERNATIVE COUNTING BLOOM FILTER

Using the same approach as for our FCF ACSM, we can construct a variation of a counting Bloom filter that we dub the d-left counting Bloom filter, or d-left CBF. The d-left CBF uses less space than a standard counting Bloom filter, with the same functionality of tracking a dynamically changing set of items under insertions and deletions. We offer a brief sketch here; the d-left CBF is more fully described in an additional paper [6]. The analysis of the d-left CBF informs why this approach is also effective in the design of ACSMs.

Again, the idea is to store a fingerprint of each item in a d-left hash table, with the intuition that d-left hashing provides a sufficiently good approximation to perfect hashing. For an insertion, we insert the fingerprint in the least loaded subtable (breaking ties to the left); for a lookup, we look for the fingerprint; and for a deletion, we remove the fingerprint. We still have the potentially problematic issue, however, that if a fingerprint associated with an items appears in multiple places, we cannot perform a deletion effectively, as we are not sure which copy of the fingerprint to delete. We avoid this problem by introducing a trick that ensures that we have never have multiple fingerprints associated with an item in the hash table.

For convenience we assume each subtable has size a power of 2, say $b = 2^z$, and we are using f bit fingerprints. First, each item is hashed using a (pseudo-)random hash function into $f + z$ bits; for an item x, call this bit string $S(x)$. We then use d (pseudo-)random permutations $h_1, h_2, \ldots, h_d$ on the string $S(x)$ of $f + z$ bits; the first f bits of $h_i(S(x))$ give the fingerprint that will be used for x in the ith subtable, and the last z bits of $h_i(S(x))$ give the index of $h_i(S(x))$ in the ith subtable. Note that the fingerprint associated with an item x can vary according to the subtable. Each cell in the table will consist of both a fingerprint and a small counter (generally 2 bits will suffice). We use the counter as follows: if, when inserting an item, we see its fingerprint already exists in a subtable, we simply increment the counter for that fingerprint, rather than insert another fingerprint in a different subtable. On deletion a counter can be decremented.

By using these permutations, we avoid the problem of ever having two fingerprints associated with an item appear in two different tables. This is because if $h_i(S(x))$ appears in the ith subtable we will never have $h_j(S(x))$ appear in the jth subtable; since $h_i(S(x))$ is in the ith subtable, if some other element y attempts to put $h_j(S(x))$ in the jth subtable, we must have that $S(x) = S(y)$ (since we are using a permutation). Hence $h_i(S(y)) = h_i(S(x))$, and instead of inserting anything into the table for item y, we will simply increment the counter associated with the fingerprint $h_i(S(x))$. Because the probability of matching fingerprints are small, a small counter (2 bits) is sufficient with very high probability. Moreover, d-left hashing offers very high utilizations of table cells, giving very good space usage.

As an example, for n items, using $d = 3$ choices, $h = 6$ cells per bucket, $b = n/12$ buckets per subtable, $f = 11$ bit fingerprints, and 2 bits per counter gives an overall cost of $(11 + 2) \cdot 3/2 = 19.5$ bits per item using a d-hash CBF. (We have used that the utilization is 2/3.) The false positive probability is approximately $n2^{-f+z} = 12(2^{-f}) \approx 0.59\%$. In contrast, a standard counting Bloom filter using 4 bits per counter (recommended to avoid overflow with high probability) and 10 counters per item uses 40 bits per item and obtains a false positive probability of about 0.82%.

The only real downside to the d-left CBF is that the hashing cannot all be done entirely in parallel; the hash function must be applied before the permutations. Also, there is perhaps slightly more work to match the fingerprint among the items in the buckets. However, the permutations can be computationally simple, the lookups remain very fast, and the space savings is certainly substantial.

Further analysis and examples can be found in [6].

5. EXPERIMENTAL EVALUATION

We divide the experimental section into two parts:

Comparing ACSM implementations: While the theoretical analysis given earlier provides considerable insight into the relative merits of our three ACSM implementations, it also has drawbacks. The theoretical models are necessarily simplified, because a complete study would need models of timers, deletions, and other effects. While simplified models (e.g., random deletion) are tenable, it seems better to use simulation to study real behavior. We provide a simulation comparison on a simple state machine in Section 5.1.

Evaluating real applications: While our simulations in Section 5.1 provide insight into whether ACSMs are *implementable* in real routers, it does not shed enough light on whether they are *useful.* In this context, the simulations have two drawbacks. First, the simulations in Section 5.1 use a contrived state machine, chosen to be sufficiently large but still simple. It is more useful to see what the figures of merit (false positive and negative probabilities, memory) are for more realistic state machines, and how much leverage one obtains by using ACSMs instead of full state machines for these applications, for a given loss of fidelity.

Second, the metrics of goodness for an ACSM (e.g., false positive rate) do not necessarily translate into application level metrics (which is what users ultimately care about) in straightforward ways. For example, in the case of the video congestion application described in Section 5.2, the relation between erroneous dropping of frames and video quality cannot easily be captured analytically. Thus, we provide a very brief study of application level performance for two applications (video congestion control and P2P identification) in Section 5.2.

5.1 Simulation

We give an example comparing performance of the various ACSMs on a simple state machine.

Experimental setup: We performed simulations using the three different algorithms to monitor $\approx 60,000$ flows with a state machine of 10 states. A simple sequential state machine was used for the experiment with synthetically generated packets for the flows. The state machine used was the simplest possible:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4.... \rightarrow 10.$$

Certain packets were encoded with state transition events.

Packet Generator: The packets for the different flows were generated using the following parameters

1. There are $n \approx 60,000$ active flows in the system.

2. Each flow is made up of $m \approx 100 \pm 40$ packets.

3. Some packets contain triggers, which correspond to state transitions. For instance a certain packet could trigger the “transition from state 3 to 4” event.

4. The packet generator sequentially generates packets, randomly selecting one of the n active flows from which to transmit. When a flow terminates, i.e., has exhausted all its packets, a new flow is created with packets containing state transition events depending on the type of the flow as explained next.

The flows are divided into three types:

1. Interesting Flows :- (30%) These flow have the correct sequence of triggers embedded in them. These triggers should lead to complete execution of the state machine from start to finish, unless there is some sort of error. The goal of the experiment is to identify these flows.

2. Noise Flows :- (30%) These flows have random triggers embedded in them. The triggers are guaranteed to not execute the state machine to completion; the final trigger from state 9 to 10 is never used.

3. Random Flows:- (Remaining) These flows have no trigger embedded in them. These flows are used to create temporal diversity in the traffic.

The packet generator was tested with all three ACSM schemes, and was allowed to run for the life of approximately 1 Million flows. The performance of the ACSM in terms of false positives/false negatives was recorded. Note that the definitions of failure probabilities are slightly different from the definitions in Section 3; here we define failure from the point of view of detecting interesting flows. The experiment records as a *false positive* cases when the ACSM finds a not-interesting (i.e., a random or noise flow) that has executed the state machine to completion. The experiment records as *a false negative* cases when the ACSM finds (at the end of the experiment) an Interesting flow whose state machine does not run to completion. If the ACSM ever returns a don't know state for a flow, the experiment records it as a *don't know* for that flow.

Algorithm Parameters: We describe the parameters we varied for each of the three ACSM implementations.

The timer task resets periodically; the period is set of about 10 times the average flow life time ($\approx$ 100 packets), multiplied by the number of active flows, for a period of 6 million packets.

Direct Bloom Filter: Each cell entry contains 2 bit for a counter plus 1 bit for a timer. The implementation used only provides for false positives and false negatives; that is, the desired state transition is given, so a don't know need never be returned. For a given memory size, the number of cells and the number of hash functions are varied to determine the optimal values for false positives and false negatives.

Stateful Bloom Filter: Each cell entry uses 4 bits for state plus 1 bit for a timer. This is sufficient for 10 states as well as a don't know and empty state. We use no counter; instead we use timers to delete entries. For a given memory size, the numbers of cells and hash functions are varied to obtain the optimal values for false positive, false negative, and don't know probabilities.

Fingerprint Compressed Filter: Again for a given memory size, the table size, the number of hash functions, cells per bucket, and fingerprint size are all varied to obtain the optimal values for false positives, false negatives, and don't know probabilities.

Simulation Results: The simulation results are summarized in Table 1. The table shows that the FCF scheme performed the best among the the three ACSM implementations, and the DBF performed the worst. Even with as small as 0.5 Mb of memory, the FCF had total error probability of under 10%, where as with about 2 Mb the total error probability was under 0.01%. By contrast, with 2Mb, the SBF had a don't know probability of 1.64% with much smaller values for the other error probabilities. While we did several other experiments that we do not describe for lack of space, we found the FCF to be uniformly superior.

5.2 The Impact of ACSMs on Application Level Performance

Earlier in the introduction, we suggested that routers and switches are likely to evolve to be more application aware. Many existing routers and switches have intelligence to monitor traffic flows for security violations and to steer traffic based on cues in packet content. This trend is likely to continue. When the corresponding applications can tolerate some loss of fidelity caused by false positives and false negatives (in return for better performance), ACSMs can be useful. This is particularly true in Application Level QoS (for example, the kind of service provided by devices such as Packeteer and P-Cube) where the network provides QoS by doing a limited amount of application level parsing to understand the relation between packets and applications, and the relative importance of each packet.

We illustrate this thesis by briefly describing and evaluating two applications: video congestion control and P2P identification. We emphasize that neither of these applications is new: there is much work that is in the literature in these two areas, but existing work uses full state machines which can be hard to implement in routers. We also emphasize that our evaluation is only meant to provide initial insight into application level benefits that accrue from using ACSMs. Much deeper and broader experimental study (which we leave to later work) is needed to fully explore the use of ACSMs in application aware networking.

5.2.1 *Video Congestion Control*

Despite failed past predictions about video over the Internet, the playing of video clips and video conferencing has increased substantially in recent years. If the history of voice is any guide, Internet video may soon be commonly used for video on demand, video conferencing, and even broadcast TV. If so, the problem of video congestion in the Internet will become significant.

McCanne's thesis [1, 14] suggested encoding video using various enhancement layers that can be discarded during times of congestion. However, this requires changes in standards and implementations. Meanwhile, video continues to flourish using the popular MPEG formats for video streaming. MPEG encodes video using 3 types of frames: I frames provide complete scene information, P frames show differential information with respect to the previous reference I or P frame, and B frames show differential information between the prior and following reference frames (the previous I or P before the B, and the next I or P after the B).

Intuitively, dropping an I frame or P frame corrupts the reference plane, making the following frames until the next I frames useless. Dropping B frames is less harmful because the following frames are not dependent on the B frame. B frames only contain temporal information and so their loss only causes motion artifacts which, up to a point, is difficult to notice. On the other hand random frame loss can cause artifacts randomly in both the temporal and spatial dimensions which are more observable at lower loss rates.

The fact that selective dropping of B frames can help during periods of congestion is well known (e.g., [9]) but its implementation in today's routers is problematic. Assuming each video stream is run over UDP, and there are several hundred thousand UDP flows concurrently through a router, to do any effective video congestion policy requires keeping state about frame boundaries for each active flow. This is because each of the I, B, and P frames are identified by a unique string at the start of the frame, frames can span

Scheme	Memory Size (bits)	False Positive	False negative	Don't Know	Other Parameters			
					num	hash		
Direct					cells	functions		
Bloom	786432	9%	19%	-	256K	3		
Filter	1572864	1.20%	4.70%	-	512K	4		
	3145728	0.03%	0.34%	-	1M	5		
					num	hash		
Stateful					cells	functions		
Bloom	524288	0.27%	5.96%	42.64%	128K	3		
Filter	1048576	0.04%	1.12%	14.45%	256K	4		
	2097152	0.00%	0.03%	1.64%	512K	5		
					table	hash	cells per	fingerprint
Fingerprint					size	functions	bucket	size (bits)
Compressed	516096	0.187%	4.278%	3.205%	6K	3	6	10
Filter	1081344	0.001%	0.011%	0.010%	8K	4	6	10
	2162688	0.000%	0.005%	0.003%	16K	4	6	18

Table 1: Simulation Results: Comparing the various ACSMs on a simple state machine.

packets, and the end of a previous frame is only signaled by the string indicating the start of the next frame.

Thus to do some form of discriminate dropping of video frames one has to maintain a small amount of state (minimally the current frame type) for all concurrent video streams. Some authors have suggested marking packets[9] to indicate priority levels in which case B frame packets can be marked as low priority. However, such marking requires standard changes (as is the case for layered encoding proposals).

Priority marking also cannot implement the full range of state machines one could envision for video congestion control. For example, one (somewhat dated) drop policy in ATM switches is "tail-dropping" which essentially converts the loss of a cell in an ATM packet to the loss of the packet [18]. A natural generalization to video is to drop all packets till the next I-frame after the loss of a significant packet within an I-frame. It does not appear to be possible to implement such a policy (which we implement using ACSMs and evaluate below) using priority marking.

To investigate the application of ACSMs to discriminate dropping of video packets within MPEG frames, we followed the approach of earlier papers in video congestion [9] and used a popular reference MPEG encoded movie "susi_040.mpg" (available from the Tektronix FTP site [22]) used by the MPEG standards committee. The file consists of 8.5% of I-Frames, 25% of P-Frames, and 66.5% of B-Frames. In terms of bytes, 29% of bytes were from I-Frames, 35% were from P-Frames, and 34% were from B-Frames.

In our experiments, we consider various drop methods; when comparing final results, we compare according to the final loss rate measured in bytes. We experimented with various dropping strategies. The first was random loss, which randomly drops frames. The second was B-frame dropping, which selectively drops B-frames randomly (which would require standards changes to do via marking as in [9])). The third policy drops packets randomly, and does pure I-frame tail dropping: once an I frame packet is dropped, all packets (and intermediate frames) are dropped till the start of the next I-frame. The fourth policy starts with B-frame dropping up to some maximum drop probability (we found that dropping more than 20% of B-frames resulted in poor quality video) followed by I-frame tail dropping.

Again, to compare the performance of the three schemes uniformly we translated the frame loss probability in the latter two cases into a byte loss probability. The first two strategies are evaluated in [9]. To the best of our knowledge, we have not seen the third or fourth strategies before. In general, we believe ACSMs can enable a much richer class of video drop policies than perhaps were thought possible.

For each strategy, we looked for two thresholds: the first was the loss threshold at which the video was almost completely unaffected by loss; the second was the threshold at which the video is severely affected and is unusable. We hasten to point that these two thresholds are perceptual, and what is more important is the broad message that discriminate video dropping can gain user satisfaction in periods of limited bandwidth than the specific numbers reported here.

More precisely, for the B-frame loss experiment we took the video file and parsed it and dropped B-frames according to some B-frame loss probability. We then played it using the Windows Media player. We gradually increased the loss probability until we first saw errors (Threshold 1) and until the quality became unacceptable (Threshold 2). We then report Threshold 1 and 2 and the loss probability, after the frame loss probability to a byte loss probability by adding the length of all the dropped frames and dividing by the total number of bytes sent in the original video.

Our results were as follows:

- **Random Drop:** Artifacts are easily seen even at a very low drop rate (i.e. 2%). Beyond 15% drop rate the picture quality is severely degraded. Thus Threshold 1 is 2% and Threshold 2 is 15%.

- **B-frame dropping:** Dropping B frames did not have any effect on the picture quality but it did have effects on the temporal plane (at very high losses, the picture seems to skip from scene to scene). In general, up 6% drop rate can be achieved without much loss in quality. The quality degrades severely at around 12%.

- **I-frame tail-dropping:** This performs worse than B-frame dropping in that artifacts are seen at 3%. However, the loss rate has to reach 25% before the picture quality gets severely degraded.

- **Combination B-frame and I-frame dropping:** We use B-frame dropping up to a loss probability of 6%, and then do I-frame tail dropping of other frames (I or P frames) for higher loss rates. This scheme performs exactly like the pure B-frame drop policy in that artifacts become noticeable at 6% but the picture gets severely degraded only at a loss rate of 30%.

In summary, the combined scheme has the best thresholds (6%, 30%). While our experiments only demonstrated the effect of loss on a *single* video stream, the use of a uniform drop probability (as in RED) will result in fairness across all video streams passing through a route. For the combined scheme, the implementation keeps a counter to track the total current B-frame byte loss probability, and instantiates the random loss of other frame packets when this counter exceeds a threshold.

As an example of the relationship between these performance numbers as ACSMs, we used an FCF ACSM with a fingerprint of 14 bits, 3 bits for state, and 1 bit for a timer to classify frame types and drop status per stream using I-frame tail dropping. To ensure no bucket overlow occurred we used a utilization of 2/3 (for the maximum number of simultaneous flows). Experimentally, the probability of incorrectly dropping a packet (from either a false positive or Don't Know) is 0.37% and the probability of not dropping a video packet that should have been dropped (from a false negative) was 0.38%. These small probabilities of error should not significantly affect the video congestion policy but the net state per stream is reduced to (14 + 3 + 1) * 1.5 = 27 bits compared to 96 + 3 + 1= 100 bits for the complete state scheme. This translates into nearly a factor of 4 reduction in memory without much impact on the video policy. Even larger savings can be obtained by increasing the false positive probability, which may be acceptable in this application.

Our results for improved quality with B-frame dropping compares with earlier results (e.g., [9]). However, [9] uses a quantitative metric called PSNR and expresses the gain as 2 to 3 db gain in PSNR using B-frame dropping, while we use perceptual loss thresholds. There are no results in [9] for I-frame tail dropping.

Our results are in no way conclusive. For example, the video we used had no audio, and the perceptual method we used could vary from user to user. However, the results do suggest the potential promise of using ACSMs for video congestion to provide a rich space of video drop policies that are implementable in routers and switches, and that do not require standards changes as in the layered encoding [1] or packet marking proposals [9].

5.2.2 *Real-time Identification of P2P Traffic*

Recent studies [10] have shown that Peer-to-Peer traffic (P2P) continues to grow to alarming percentages, such as 30% of all traffic at peering points. Most organizations and even ISPs would like to rate control such traffic in favor of other possibly more mission-critical, traffic. Unfortunately, as part of the battle with such legal entities as the RIAA, P2P traffic constantly evolves to conceal itself. P2P traffic routinely tunnels in on well known ports such as Port 80, and many routinely operate on any port number. Thus simple discrimination of P2P packets based on port numbers is no longer very useful. A recent proposal [19] suggests looking for content signatures within packets (e.g., "GNUTELLA CONNECT") to identify P2P packets. Unfortunately, motivated by legal sticks and financial carrots, P2P protocols have shown a growing trend to obfuscate their payloads using encryption.

As a more compelling approach, recent work in [10] shows that there are certain traffic flow patterns, detectable by simple state machines, that have a high degree of accuracy in identifying P2P flows. For example, [10] shows that a (DestIP, SourceIP) pair that has a concurrent UDP connection (often used for control queries and replies) and TCP connection (used for file transfer) has a high probability of being P2P traffic. The work also uses patterns of (IP, port) pairs for another test.

We believe that such traffic characterization based on traffic flow analysis (for P2P or other uses such as security) can benefit from the use of ACSMs. The paper [10] uses extensive (and time-consuming) *offline* trace analysis. This is useful for traffic characterization after the fact, and can be used with tools such as NetFlow to provide fairly slow analysis of P2P traffic. However, ACSMs can be used for online, real-time characterization using the same techniques and can thus be used for real-time control (using say rate limiting). The lack of fidelity of ACSMs due to false positives and negatives is not a large impediment, because the very heuristics used to identify P2P traffic also have some probability of error that is often significantly larger than the errors caused by using ACSMs.

To quickly understand how P2P identification can benefit from ACSMs we simulated the simplest state machine test in [10] (TCP-UDP pairs). Since we could not obtain traces of the traffic used in [10], we created a synthetic log containing 64K active flows (a flow is timed out after 64 seconds of inactivity). The flows (based on the data in [10]) have an average of 15 packets per flow. We use a parameter we call the P2P percentage that controls what fraction of the TCP flows have a concurrent UDP flow. Between the start and end of these selected flows, we randomly send a UDP packet. The experiment compares, for different values of the P2P percentage parameter, the number of flows misclassified as P2P using an ACSM compared to that done by a full state machine. (We are assuming that the full state machine is perfectly correct, but in practice it is not [10].)

We present results using an FCF ACSM with 2^{11} cells, 4 hash functions, a fingerprint Size of 14 Bits, and 9 entries per cell. Thus the total memory used was approximately 1 Mbit (more precisely, $2^{11} * 4 * 9 * (14 + 2) = 1179648$ bits). The results were as follows:

- **15 % P2P Traffic:** 0% False positive, 0.29% False Negative, and 0.17% DK
- **25 % P2P traffic:** 0% False positive, 0.31% False Negative, 0.156% DK
- **40 % P2P traffic:** 0% False positive, 0.34% False Negative, 0.149% DK

Even if we add all these probabilities together (as the probability of misclassification), the probabilities compare very well with the misclassification probabilities reported in [10] for the heuristics themselves (for example, false positive rates of 6%). On the other hand, the slight loss in fidelity is compensated by a very large factor reduction in memory (from 96 bits per flow in the naive scheme to 18 bits per flow using ACSMs). Better still, this can be done with a few Mbits of memory which is implementable by using on-chip SRAM even at 20 Gbps and higher speeds.

There are several limitations to our study, however. First, we did not use the actual traces and several interesting dynamics of real traffic (timeouts, Zipf flow sizes etc.) are not captured by our synthetic model. Rather than improve our synthetic model, we plan to work on the real traces once we obtain them. Second, we did not implement the full state machine in [10]. Third, our model of concurrent TCP and UDP connections only included a UDP connection starting within a TCP connection, not vice versa. While these extensions are necessary in a deeper, trace-drive study, the initial results strongly suggest the potential promise of ACSMs for real-time P2P identification and control.

In closing, we note that P2P identification, like security, is likely to be an arms race. If these heuristics get widely used, it is likely that P2P authors will change their programs to invalidate these detection methods. However, if a router allows programmable ACSMs, the router can be reprogrammed to employ new detection state machines, much in the way virus definitions are updated when new threats are discovered.

6. CONCLUSION

In this paper, we have introduced Approximate Concurrent State Machines. While similar in spirit to Bloom filters, our best scheme is based on a combination of hashing and fingerprints, using d-left hashing to obtain a near-perfect hash function in a dynamic setting. Somewhat surprisingly, when we specialize to the case of membership checking in a dynamic set, we find that our data structure takes much less space than a comparable counting Bloom filter.

ACSMs are particularly relevant to an increasing trend in networking devices to be more application aware. This is often done by keeping a state machine for each TCP flow. Because of the great memory needs (1 million concurrent connections, for example, is the standard for all IDS devices in the market), these implementations often resort to low speed DRAM and hence are often implemented at speeds of a few Gbps or less. By reducing the memory needs by a factor of 5 or more, ACSMs allow the potential for on-chip SRAM and hence higher speed implementations of such application-aware network devices. Finally, the loss in fidelity caused by small probabilities of error are often tolerable for Application Level QoS mechanisms for which an error only means stepping up (or down) the QoS level for the occasional connection.

We have sketched the potential uses of ACSMs for application-aware forwarding using two specific examples: Active Queue Management schemes for video flows, and real-time detection and control of P2P traffic. We also believe programmable ACSMs can be a powerful tool for traffic analysis. Our investigation of applications is preliminary, and more sophisticated and detailed experiments are needed to confirm the promise of ACSMs for applications. This paper, instead, focuses on the introduction of the problem, the introduction of 3 specific data structures to solve the problem, and the analytical and simulation comparison of the 3 schemes. Given that Bloom filters and related variants have found many applications, we believe that ACSMs will also find many new applications in the future.

7. REFERENCES

[1] E. Amir, S. McCanne. M. Vitterli. A Layered DCT coder for Internet Video. In *Proc. IEEE International Conference on Image Processing*, Lausanne, Switzerland, Sept 1996.

[2] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.

[3] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.

[4] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp 30-39, 2004.

[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. To appear in the 2006 European Symposium on Algorithms.

[7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, Stanford, CA, August 2003. IEEE Computer Society Press.

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

[9] D. Forsgren, U. Jennehag. P. Osterberg, Objective Endtoend QoS Gain from Packet Prioritization and Layering in MPEG-2 streaming video. At http://amp.ece.cmu.edu/packetvideo2002/papers/61-ananhseors.pdf

[10] T. Karargiannis, A. Broido, M. Faloutsos, and K.C. Claffy. Transport Layer Identification of P2P Traffic. In *Proceedings of ACM SIGCOMM*, 2004.

[11] T. Karargiannis, A. Broido, M. Faloutsos, and K.C. Claffy. BLINC: Multilevel Traffic Classification in the Dark. In *Proceedings of ACM SIGCOMM*, 2005.

[12] A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Multiple Choices. In *Proc. of the Forty-Third Annual Allerton Conference*, 2005.

[13] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom Filters: Design Innovations and Novel Applications. In *Proc. of the Forty-Third Annual Allerton Conference*, 2005.

[14] Steve McCanne. Scalable Compression and Transmission of Internet Multicast Video. Ph. D. Thesis, Berkeley

[15] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.

[16] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[17] A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 823-829, 2005.

[18] A. Romanow and S. Floyd. Dynamics of TCP Traffic over ATM Networks. *IEEE Journal on Selected Areas in Communications*, 13(4): 633-641 (1995).

[19] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-network identification of P2P Traffic Using Application Signatures. In *13th International World Wide Web Conference*, New York City, 17-22 May 2004.

[20] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: an aid to network processing. In *Proceedings of ACM SIGCOMM*, pp. 181-192, 2005.

[21] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, B. Schwartz, S. Kent, and W. Strayer. Single-Packet IP Traceback. *IEEE/ACM Transactions on Networking*, 10(6):721-734, 2002.

[22] Tektronix FTP site, ftp://ftp.tek.com/tv/test/streams/Element/MPEG-Video/625/

[23] K. Thomson, G. J. Miller, and R. Wilder. Wide-area traffic patterns and characteristics. *IEEE Network*, December 1997.

[24] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the* 40^{th} *IEEE-FOCS*, pp. 131-140, 1999.