Heavy-Hitter Detection Entirely in the Data Plane

Vibhaalakshmi Sivaraman
Princeton University

Srinivas Narayana
MIT CSAIL

Ori Rottenstreich
Princeton University

S. Muthukrishnan
Rutgers University

Jennifer Rexford
Princeton University

ABSTRACT

Identifying the "heavy hitter" flows or flows with large traffic volumes in the data plane is important for several applications *e.g.,* flow-size aware routing, DoS detection, and traffic engineering. However, measurement in the data plane is constrained by the need for line-rate processing (at 10-100Gb/s) and limited memory in switching hardware. We propose HashPipe, a heavy hitter detection algorithm using emerging programmable data planes. HashPipe implements a pipeline of hash tables which retain counters for heavy flows while evicting lighter flows over time. We prototype HashPipe in P4 and evaluate it with packet traces from an ISP backbone link and a data center. On the ISP trace (which contains over 400,000 flows), we find that HashPipe identifies 95% of the 300 heaviest flows with less than 80KB of memory.

KEYWORDS

Software-Defined Networks; Network Monitoring; Programmable Networks; Network Algorithms

1 INTRODUCTION

Many network management applications can benefit from finding the set of flows contributing significant amounts of traffic to a link: for example, to relieve link congestion [5], to plan network capacity [18], to detect network anomalies and attacks [23], or to cache forwarding table entries [36]. Further, identifying such "heavy hitters" at small time scales (comparable to traffic variations [1, 5]) can enable dynamic routing of heavy flows [16, 35] and dynamic flow scheduling [41].

It is desirable to run heavy-hitter monitoring at all switches in the network all the time, to respond quickly to short-term traffic variations. *Can packets belonging to heavy flows be identified as the packets are processed in the switch*, so that switches may treat them specially?

Existing approaches to monitoring heavy items make it hard to achieve reasonable accuracy at acceptable overheads (§2.2). While packet *sampling* in the form of NetFlow [12] is widely deployed, the CPU and bandwidth overheads of processing sampled packets in software make it infeasible to sample at sufficiently high rates (sampling just 1 in 1000 packets is common in practice [34]). An alternative is to use sketches, *e.g.,* [14, 24, 25, 45] that *hash* and *count* all packets in switch hardware. However, these systems incur a large memory overhead to retrieve the heavy hitters — ideally, we wish to use memory proportional to the number of the heavy flows (say the top hundred). There may be tens of thousands of active flows any minute on an ISP backbone link (§5) or a data center top-of-rack switch [4].

Emerging programmable switches [3, 7, 9] allow us to do more than sample, hash, and count, which suggests opportunities to run novel algorithms on switches. While running at line rates of 10-100 Gbps per port over 10-100 ports, these switches can be programmed to maintain state over multiple packets, *e.g.,* to keep flow identifiers as keys along with counts. Stateful manipulations can also be *pipelined* over multiple stages, with the results carried by the packet from one stage to the next and compared to stored state in subsequent stages.

However, switches are also constrained by the need to maintain high packet-processing throughput, having:

- a deterministic, small time budget (1 ns [7]) to manipulate state and process packets at each stage;
- a limited number of accesses to memory storing state at each stage (typically just one read-modify-write);
- a limited amount of memory available per stage (*e.g.,* 1.4MB shared across forwarding and monitoring [7]);

SOSR '17, Santa Clara, CA

DOI: 10.1145/3050220.3063772

- a need to move most packets just once through the pipeline, to avoid stalls and reduced throughput ("feed-forward" [19]).

We present HashPipe, an algorithm to track the k heaviest flows with high accuracy (§3) within the features and constraints of programmable switches. HashPipe maintains both the flow identifiers ("keys") and counts of heavy flows in the switch, in a pipeline of hash tables. When a packet hashes to a location in the first stage of the pipeline, its counter is updated (or initialized) if there is a *hit* (or an empty slot). If there is a *miss*, the new key is inserted at the expense of the existing key. In all downstream stages, the key and count of the item just evicted are carried along with the packet. The carried key is looked up in the current stage's hash table. Between the key looked up in the hash table and the one carried, the key with the *larger count* is retained in the hash table, while the other is either carried along with the packet to the next stage, or totally removed from the switch if the packet is at the last stage. Hence, HashPipe "smokes out" the heavy keys within the limited available memory, using pipelined operation to sample multiple locations in the hash tables, and evicting lighter keys in favor of heavier keys, with updates to exactly one location per stage.

We prototype HashPipe in P4 [6] (§4) and test it on the public-domain behavioral switch model [32]. We evaluate HashPipe with packet traces obtained from an ISP backbone link (from CAIDA) and a data center, together containing over 500 million packets. We show that HashPipe can provide high accuracy (§5). In the backbone link trace, HashPipe incurs less than 5% false negatives and 0.001% false positives when reporting 300 heavy hitters (keyed by transport five-tuple) with just 4500 counters (less than 80KB) overall, while the trace itself contains 400,000 flows. The errors both in false negatives and flow count estimations are lower among the heavier flows in the top k. At 80KB of memory, HashPipe has 15% lower false negatives with respect to sample and hold [17], and 3-4% with respect to an enhanced version of the count-min sketch [14].

2 BACKGROUND ON HEAVY-HITTER DETECTION

2.1 Problem Formulation

Heavy hitters. "Heavy hitters" can refer to all flows that are larger (in number of packets or bytes) than a fraction t of the total packets seen on the link (the "threshold-t" problem). Alternatively, the heavy hitters can be the top k flows by size (the "top-k" problem). Through the rest of this paper, we use the "top-k" definition.

Flow granularity. Flows can be defined at various levels of granularity, such as IP address (*i.e.,* host), transport port number (i.e., application), or five-tuple (*i.e.,* transport connection). With a finer-grained notion of flows, the size and number of keys grows, requiring more bits to represent the key and more entries in the data structure to track the heavy flows accurately.

Accuracy. A technique for detecting heavy hitters may have false positives (*i.e.,* reporting a non-heavy flow as heavy), false negatives (*i.e.,* not reporting a heavy flow), or an error in estimating the sizes of heavy flows. The impact of errors depends on the application. For example, if the switch performs load balancing in the data plane, a few false negatives may be acceptable, especially if those heavy flows can be detected in the next time interval. As another example, if the network treats heavy flows as suspected denial-of-service attacks, false positives could lead to unnecessary alarms that overwhelm network operators. When comparing approaches in our evaluations, we consider all three metrics.

Overhead. The overhead on the switch includes the total amount of memory for the data structure, the number of matching stages used in the switch pipeline (constrained by a switch-specific maximum, say 16 [7]). The algorithms are constrained by the nature and amount of memory access and computation per match stage (*e.g.,* computing hash functions). On high-speed links, the number of active five-tuple flows per minute can easily be in the tens of thousands, if not more. Our central goal is to maintain data-plane state that is proportional to the target number k of heavy hitters (*e.g.,* $5k$ or $10k$), rather than the number of active flows. In addition, we would like to use as few pipeline stages as possible, since the switch also needs to support other functionality related to packet forwarding and access control.

2.2 Existing Solutions

The problem of finding heavy flows in a network is an instance of the "frequent items" problem, which is extremely well studied in the streaming algorithms literature [13]. While high accuracy and low overhead are essential to any heavy hitter detection approach, we are also specifically interested in implementing these algorithms within the constraints of emerging programmable switches (§1). We classify the approaches into two broad categories, *sampling* and *streaming*, discussed below.

Packet sampling using NetFlow [12] and sFlow [38] is commonly implemented in routers today. These technologies record a subset of network packets by sampling, and send the sampled records to collectors for analysis. To keep packet processing overheads and data collection bandwidth low, NetFlow configurations in practice use aggressively low sampling probabilities, *e.g.,* 1% or

even 0.01% [30, 34]. Such undersampling can impact the estimation accuracy.

Sample and hold [17] enhances the accuracy of packet sampling by keeping counters for *all* packets of a flow in a *flow table,* once a packet from that flow is sampled. Designing large flow tables[1] for fast packet *lookup* is well-understood: hash table implementations are already common in switches, for example in router forwarding tables and NetFlow [24]. However, it is challenging to handle hash collisions when *adding* flows to the flow table, when a packet from a new flow is sampled. Some custom hardware solutions combine the hash table with a "stash" that contains the overflow, *i.e.,* colliding entries, from the hash table [22]. But this introduces complexity in the switch pipeline, and typically involves the control plane to add entries into the stash memory. Ignoring such complexities, we evaluate sample and hold in §5 by liberally allowing the flow table to lookup packets *anywhere* in a *list* of a bounded size.

Streaming algorithms implement data structures with bounded memory size and processing time per packet, while processing *every packet* in a large stream of packets in one pass. The algorithms are designed with provable accuracy-memory tradeoffs for specific statistics of interest over the packets. While these features make the algorithms attractive for network monitoring, the specific algorithmic operations on each packet determine feasibility on switch hardware.

Sketching algorithms like count-min sketch [14] and count sketch [10] use per-packet operations such as hashing on packet headers, incrementing counters at hashed locations, and determining the minimum or median among a small number of the counters that were hashed to. These operations can all be efficiently implemented on switch hardware [45]. However, these algorithms do not track the flow identifiers of packets, and hash collisions make it challenging to "invert" the sketch into the constituent flows and counters. Simple workarounds like collecting packets that have high flow count estimates in the sketch could result in significant bandwidth overheads, since most packets from heavy flows will have high estimates.

Techniques like group testing [15], reversible sketches [37], and FlowRadar [24] can decode keys from hash-based sketches. However, it is challenging to read off an accurate counter value for a packet in the switch pipeline itself since the decoding happens off the fast packet-processing path.

Counter-based algorithms [27, 29] focus on measuring the heavy items, maintaining a table of flows and corresponding counts. They employ per-counter increment and subtraction operations, but potentially all counters in the table are updated during some flow insertions. Updating multiple counters in a single stage is challenging within the deterministic time budget for each packet.

Space saving is a counter-based algorithm that uses only $O(k)$ counters to track k heavy flows, achieving the lowest memory usage possible for a fixed accuracy among deterministic heavy-hitter algorithms—both theoretically [28] and empirically [13]. Space saving only updates one counter per packet, but requires finding the item with the minimum counter value in the table. Unfortunately, scanning the entire table on each packet, or finding the minimum in a table efficiently, is not directly supported on emerging programmable hardware (§1). Further, maintaining data structures like sorted linked lists [28] or priority queues [41] requires multiple memory accesses within the per-packet time budget. However, as we show in §3, we are able to adapt the key ideas of the space saving algorithm and combine it with the functionality of emerging switch hardware to develop an effective heavy hitter algorithm.

[1]Large exact-match lookups are typically built with SRAM, as large TCAMs are expensive.

3 HASHPIPE ALGORITHM

As described in §2.2, HashPipe is heavily inspired by the space saving algorithm [28], which we describe now (§3.1). In the later subsections (§3.2-§3.4), we describe our modifications to the algorithm to make it amenable to switch implementation.

3.1 Space Saving Algorithm

To track the k heaviest items, space saving uses a table with m (which is $O(k)$) slots, each of which identifies a distinct flow key and its counter. All slots are initially empty, *i.e.,* keys and counters are set to zero.

Algorithm 1: Space Saving algorithm [28]

▷ Table T has m slots, either containing (key_j, val_j) at slot $j \in \{1, \ldots, m\}$, or empty. Incoming packet has key $iKey$.
if $\exists$ *slot* j *in* T *with* $iKey = key_j$ **then**
  $val_j \leftarrow val_j + 1$
else
  if $\exists$ *empty slot* j *in* T **then**
    $(key_j, val_j) \leftarrow (iKey, 1)$
  else
    $r \leftarrow argmin_{j \in \{1,\ldots,m\}}(val_j)$
    $(key_r, val_r) \leftarrow (iKey, val_r + 1)$
  end
end

The algorithm is summarized in Algorithm 1.[2] When a packet arrives, if its corresponding flow isn't already in the table, and there is space left in the table, space saving inserts the new flow with a count of 1. If the flow is present in the table, the algorithm updates the corresponding flow counter. However, if the table is full, and the flow isn't found in the table, the algorithm replaces the flow entry that has the *minimum* counter value in the table with the incoming packet's flow, and increments this minimum-valued counter.

This algorithm has three useful accuracy properties [28] that we list here. Suppose the true count of flow key_j is c_j, and its count in the table is val_j. First, no flow counter in the table is ever underestimated, *i.e.,* $val_j \geq c_j$. Second, the minimum value in the table val_r is an upper bound on the overestimation error of any counter, *i.e.,* $val_j \leq c_j + val_r$. Finally, any flow with true count higher than the average table count, *i.e.,* $c_j > C/m$, will always be present in the table (here C is the total packet count added into the table). This last error guarantee is particularly useful for the threshold-heavy-hitter problem (§2.1), since by using $1/t$ counters, we can extract all flows whose true count exceeds a threshold t of the total count. However, this guarantee cannot be extended directly to the top-k problem, since due to the heavy-tailed nature of traffic [17], the kth heaviest flow contributes nowhere close to $1/k$ of the total traffic.

The operation of finding the minimum counter in the table (line 8)—possibly for each packet—is difficult within switch hardware constraints (§2.2). In the following subsections, we discuss how we incrementally modify the space saving algorithm to run on switches.

3.2 Sampling for the Minimum Value

The first simplification we make is to look at the minimum of a small, *constant* number d of randomly chosen counters, instead of the entire table (in line 8, Algorithm 1). This restricts the worst-case number of memory accesses per packet to a small fixed constant d. The modified version, HashParallel, is shown in Algorithm 2. The main change from Algorithm 1 is in the set of table slots examined (while looking up or updating any key), which is now a set of d slots obtained by hashing the incoming key using d independent hash functions.

In effect, we *sample* the table to estimate a minimum using a few locations. However, the minimum of just d slots can be far from the minimum of the entire table of m slots. An inflated value of the minimum could impact the useful error guarantees of space saving (§3.1). Our evaluations in §5.4 show that the distributions of the minimum of the entire table and of the subsample are comparable.

However, Algorithm 2 still requires the switch to read d locations at once to determine their minimum, and then write back the updated value. This necessitates a read-modify-write operation, involving d reads and one write *anywhere in table memory*, within the per-packet time budget. However, multiple reads to the same table ($d > 1$) are supported neither in switch programming languages today [6] nor in emerging programmable switching chips [7]. Further, supporting multiple reads for every packet at line rate requires multiported memories with strict access-latency guarantees, which can be quite expensive in terms of area and power [20, 44].

3.3 Multiple Stages of Hash Tables

The next step is to reduce the number of reads to memory to facilitate operation at line rate. We split the counter table T into d disjoint tables, and we read exactly one slot per table. The algorithm is exactly the same as Algorithm 2; except that now hash function h_i returns only slots in the ith stage.

This design enables pipelining the memory accesses to the d tables, since different packets can access different tables at the same time. However, packets may need to make two passes through this pipeline: once to determine the counter with the minimum value among d slots, and a second time to update that counter. The second pass is possible through "recirculation" of packets through the switching pipeline [2, 39, 42] with additional metadata on the packet, allowing the switch to increment the minimum-valued counter in the second pass. However, the second pass is potentially needed for *every packet*, and recirculating every packet can halve the pipeline bandwidth available to process packets.

Algorithm 2: HashParallel: Sample d slots at once

```
1  ▷ Hash functions h_i(iKey) → slots, i ∈ {1, ..., d}
2  H = {h_1(iKey), ..., h_d(iKey)}
3  if ∃ slot j ∈ H with iKey = key_j then
4  |   val_j ← val_j + 1
5  else
6  |   if ∃ empty slot j ∈ H then
7  |   |   (key_j, val_j) ← (iKey, 1)
8  |   else
9  |   |   r ← argmin_{j∈H}(val_j)
10 |   |   (key_r, val_r) ← (iKey, val_r + 1)
11 |   end
12 end
```

[2] We show the algorithm for packet counting; it easily generalizes to byte counts.

3.4 Feed-Forward Packet Processing

We now alleviate the need to process a packet more than once through the switch pipeline, using two key ideas.

Track a rolling minimum. We track the minimum counter value seen so far (and its key) as the packet traverses the pipeline, by *moving* the counter and key through the pipeline as *packet metadata.* Emerging programmable switches allow the use of such metadata to communicate results of packet processing between different stages, and such metadata can be written to at any stage, and used for packet matching at a later stage [42].

Algorithm 3: HashPipe: Pipeline of d hash tables

```
▷ Insert in the first stage
l_1 ← h_1(iKey)
if key_{l_1} = iKey then
    val_{l_1} ← val_{l_1} + 1
    end processing
end
else if l_1 is an empty slot then
    (key_{l_1}, val_{l_1}) ← (iKey, 1)
    end processing
end
else
    (cKey, cVal) ← (key_{l_1}, val_{l_1})
    (key_{l_1}, val_{l_1}) ← (iKey, 1)
end
▷ Track a rolling minimum
for i ← 2 to d do
    l ← h_i(cKey)
    if key_l = cKey then
        val_l ← val_l + cVal
        end processing
    end
    else if l is an empty slot then
        (key_l, val_l) ← (cKey, CVal)
        end processing
    end
    else if val_l < cVal then
        swap (cKey, cVal) with (key_l, val_l)
    end
end
```

As the packet moves through the pipeline, the switch hashes into each stage on the *carried key,* instead of hashing on the key corresponding to the incoming packet. If the keys match in the table, or the slot is empty, the counter is updated in the usual way, and the key needs no longer to be carried forward with the packet. Otherwise, the keys and counts corresponding to the *larger* of the counters that is carried and the one in the slot is written back into the table, and the smaller of the two is carried on the packet. We leverage arithmetic and logical action operations available in the match-action tables in emerging switches [7] to implement the counter comparison. The key may be carried to the next stage, or evicted completely from the tables when the packet reaches the last (*i.e., d*th) stage.

Always insert in the first stage. If the incoming key isn't found in the first stage in the pipeline, there is no associated counter value to compare with the key that is in that table. Here, we choose to *always insert the new flow* in the first stage, and evict the existing key and counter into packet metadata. After this stage, the packet can track the rolling minimum of the *subsequent stages* in the usual way described above. The final algorithm, HashPipe, is shown in Algorithm 3.

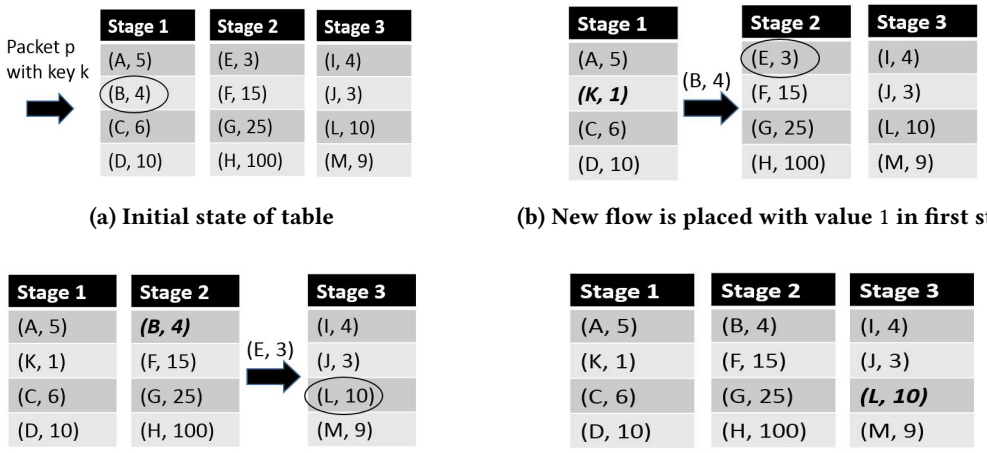
One consequence of always inserting an incoming key in the first stage is the possibility of duplicate keys across different tables in the pipeline, since the key can exist at a later stage in the pipeline. Note that this is unavoidable when packets only move once through the pipeline. It is possible that such duplicates may occupy space in the table, leaving fewer slots for heavy flows, and causing evictions of heavy flows whose counts may be split across the duplicate entries.

However, many of these duplicates are easily *merged* through the algorithm itself, *i.e.,* the minimum tracking merges counters when the carried key has a "hit" in the table. Further, switches can easily estimate the flow count corresponding to any packet in the data plane itself by summing all the matching flow counters; so can a data collector, after reading the tables out of the switch. We also show in our evaluations (§5.1) that duplicates only occupy a small portion of the table memory.

Fig. 1 illustrates an example of processing a packet using HashPipe. A packet with a key K enters the switch pipeline (a), and since it isn't found in the first table, it is inserted there (b). Key B (that was in the slot currently occupied by K) is carried with the packet to the next stage, where it hashes to the slot containing key E. But since the count of B is larger than that of E, B is written to the table and E is carried out on the packet instead (c). Finally, since the count of L (that E hashes to) is larger than that of E, L stays in the table (d). The net effect is that the new key K is inserted in the table, and the minimum of the three keys B, E, and L—namely E—is evicted in its favor.

4 HASHPIPE PROTOTYPE IN P4

We built a prototype of HashPipe using P4 version 1.1 [42]. We verified that our prototype produced the same results as our HashPipe simulator by running a small number of



(a) Initial state of table

(b) New flow is placed with value 1 in first stage

(c) B being larger evicts E

(d) L being larger is retained in the table

Figure 1: An illustration of HashPipe.

```
1   action doStage1(){
2     mKeyCarried = ipv4.srcAddr;
3     mCountCarried = 0;
4     modify_field_with_hash_based_offset(mIndex, 0,
          stage1Hash, 32);
5
6     // read the key and value at that location
7     mKeyTable = flowTracker[mIndex];
8     mCountTable = packetCount[mIndex];
9     mValid = validBit[mIndex];
10
11    // check for empty location or different key
12    mKeyTable = (mValid == 0) ? mKeyCarried : mKeyTable;
13    mDif = (mValid == 0) ? 0 : mKeyTable - mKeyCarried;
14
15    // update hash table
16    flowTracker[mIndex] = ipv4.srcAddr;
17    packetCount[mIndex] = (mDif == 0) ? mCountTable+1: 1;
18    validBit[mIndex] = 1;
19
20    // update metadata carried to the next table stage
21    mKeyCarried = (mDif == 0) ? 0 : mKeyTable;
22    mCountCarried = (mDif == 0) ? 0 : mCountTable;
23  }
24
```

Listing 1: HashPipe stage with insertion of new flow. Fields prefixed with m are metadata fields.

artificially generated packets on the switch behavioral model [32] as well as our simulator, and ensuring that the hash table is identical in both cases at the end of the measurement interval.

At a high level, HashPipe uses a match-action stage in the switch pipeline for each hash table. In our algorithm, each match-action stage has a single default action—the algorithm execution—that applies to every packet. Every stage uses its own P4 *register arrays*—stateful memory that persists across successive packets—for the hash table. The register arrays maintain the flow identifiers and associated counts. The P4 action blocks for the first two stages are presented in Listings 1 and 2; actions for further stages are identical to that of stage 2. The remainder of this section walks through the P4 language constructs with specific references to their usage in our HashPipe prototype.

Hashing to sample locations: The first step of the action is to hash on the flow identifier (source IP address in the listing) with a custom hash function, as indicated in line 4 of Listing 1. The result is used to pick the location where we check for the key. The P4 behavioral model [32] allows customized hash function definitions. We use hash functions of the type $h_i = (a_i \cdot x + b_i)\%p$ where the chosen a_i, b_i are co-prime to ensure independence of the hash functions across stages. Hash functions of this sort are implementable on hardware and have been used in prior work [24, 25].

Registers to read and write flow statistics: The flows are tracked and updated using three registers: one to track the flow identifiers, one for the packet count, and one to test validity of each table index. The result of the hash function is used to index into the registers for reads and writes. Register reads occur in lines 6-9 and register writes occur in lines 15-18 of Listing 1. When a flow identifier is read from the register, it is checked against the flow identifier currently carried. Depending on whether there is a match, either the value 1 is written back or the current value is incremented by 1.

Packet metadata for tracking current minimum: The values read from the registers are placed in packet metadata since we cannot test conditions directly on the register values in P4. This enables us to compute

```
1   action doStage2{
2    ...
3    mKeyToWrite = (mCountInTable < mCountCarried) ?
         mKeyCarried : mKeyTable));
4    flowTracker[mIndex] = (mDif == 0) ? mKeyTable :
         mKeyToWrite;
5
6    mCountToWrite = (mCountTable < mCountCarried) ?
         mCountCarried : mCountTable;
7    packetCount[mIndex] = (mDif == 0) ? (mCountTable +
         mCountCarried) : mCountToWrite;
8
9    mBitToWrite = (mKeyCarried == 0) ? 0 : 1);
10   validBit [mIndex] = (mValid == 0) ? mBitToWrite : 1);
11   ...
12  }
```

Listing 2: HashPipe stage with rolling minimum. Fields prefixed with m are metadata fields.

the minimum of the carried key and the key in the table before writing back into the register (lines 11-13 of Listing 1 and lines 3, 6, and 9 of Listing 2). Packet metadata also plays a crucial role in conveying state (the current minimum flow id and count, in this case) from one stage to another. The metadata is later used to compute the sample minimum. The updates that set these metadata across stages are similar to lines 20-22 of Listing 1.

Conditional state updates to retain heavier flows: The first pipeline stage involves a conditional update to a register to distinguish a hit and a miss for the incoming packet key in the table (line 17, Listing 1). Subsequent stages must also write back different values to the table key and count registers, depending on the result of the lookup (hit/miss) and a comparison of the flow counts. Accordingly, we perform a conditional write into the flow id and count registers (lines 4 and 7, Listing 2). Such conditional state updates are feasible at line rate [19, 40].

5 EVALUATION

We now evaluate HashPipe through trace-driven simulations. We tune the main parameter of HashPipe—the number of table stages d—in §5.1. We evaluate the performance of HashPipe in isolation in §5.2, and then compare it to prior sampling and sketching solutions in §5.3. Then, we examine the performance of HashPipe in context of the idealized algorithms it is derived from (§3) in §5.4.

Experiment setup. We compute the k heaviest flows using two sets of traces. The first trace is from a 10Gb/s ISP backbone link, recorded in 2016 and available from CAIDA [8]. We measure heavy hitters aggregated by transport 5-tuple. The traffic trace is 17 minutes long, and contains 400 million packets. We split this trace into 50 chunks, each being about 20 seconds long, with 10 million packets. The chunks on average contain about 400,000 5-tuple flows each. Each chunk is one trial, and each data point in the graphs for the ISP trace reports the average across 50 trials. We assume that the switch zeroes out its tables at the end of each trial, which corresponds to a 20 second "table flush" period.

The second trace, recorded in 2010, is from a data center [4] and consists of about 100 million packets in total. We measure heavy hitters aggregated by source and destination IPs. We split the trace into 1 second intervals corresponding to the time scale at which data center traffic exhibits stability [5].

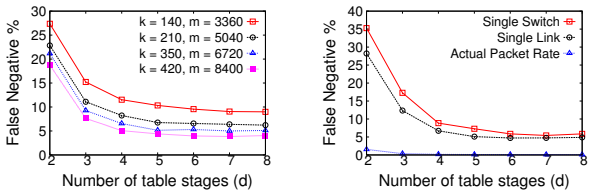
The data center trace is two and a half hours long, with roughly 10K packets (300 flows) per second. We additionally replay the trace at two higher packet rates to test whether HashPipe can provide good accuracy over the 1 second time scale. We assume a "typical" average packet size of 850 bytes and a 30% network utilization as reported in prior traffic studies [4]. For a switch clocked at 1GHz with 48 ports of 10Gb/s each, this corresponds to roughly 20 million packets per second through the entire switch, and 410K packets per second through a single link. At these packet rates, the trace contains 20,000 and 3200 flows per second respectively. For each packet rate, we report results averaged from multiple trials of 1 second each.

Metrics. As discussed in §2.1, we evaluate schemes on false negatives (% heavy flows that are not reported), false positives (% non-heavy flows that are reported), and the count estimation error (% error for heavy flows). Note that for the top-k problem, the false positive error is just a scaled version of the false negative.

5.1 Tuning HashPipe

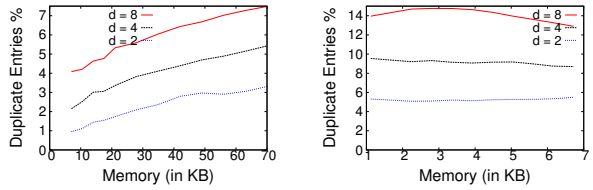
Given a total memory size m, HashPipe's only tunable parameter is the number of table stages d that it uses. Once d is fixed, we simply partition the available memory equally into d hash tables. As d increases, the number of table slots over which a minimum is computed increases, leading to increased retention of heavier keys. However, with a fixed total memory, an increase in the value of d decreases the per-stage hash table size, increasing the likelihood of hash collisions, and also of duplicates (§3.4). The switch hardware constrains the number of table stages to a small number, *e.g.,* 6-16 [7, 31].

Fig. 2 shows the impact of changing d on the false negatives. For the ISP trace, we plot false negatives for different sizes of memory m and different number of desired heavy hitters k. As expected, the false negatives reduce as d increases starting at $d = 2$, but the decrease quickly tapers off in the range between 5–8, across the different (m, k) curves. For the data center trace, we show false negatives for $k = 350$ with a memory of 840



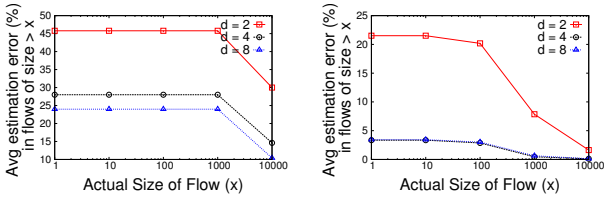
(a) ISP backbone (b) Data center

Figure 2: Impact of table stages (d) on false negatives. Error decreases as d grows, and then flattens out.



(a) ISP backbone (b) Data center (link)

Figure 3: Prevalence of duplicate keys in tables. For different d values under the memory range tested, the proportion of duplicates is between 5-10% for the ISP trace and 5-15% for the data center trace (link packet rate).



(a) ISP backbone (b) Data center (link)

Figure 4: Average estimation error (%) of flows whose true counts are higher than the x-value in the graph. Error decreases as d grows but the benefit diminishes with increasing d. Error decreases with actual flow size.

counters (15KB), and we see a similar trend across all three packet rates. The false positives, elided here, also follow the same trend.

To understand whether duplicates impact the false negative rates, we also show the prevalence of duplicates in HashPipe's hash tables in Fig. 3. Overall, duplicates only take up between 5-10% of the available table size in the ISP case and between 5-14% in the data center case. As expected, in going from $d = 2$ and $d = 8$, the prevalence of duplicates in HashPipe's table increases.

Fig. 4 shows the count estimation error (as a percentage of actual flow size) *for flows in HashPipe's tables* at the end of each measurement interval, with a memory size of 640 counters (11.2KB). In general, the error reduces as d increases, but the reduction from $d = 4$ to $d = 8$ is less significant than the reduction from $d = 2$ to $d = 4$. In the ISP trace, the estimation error is stable across flow sizes since most flows recorded by HashPipe have sizes of at least 1000. In the data center trace where there are fewer total flows, there is a more apparent decrease in error with true flow size, with flows of size $x > 1000$ having near-perfect count estimations.

Choosing $d = 6$ table stages. To summarize, we find that (i) as the number of table stages increases above $d = 4$, all the accuracy metrics improve; (ii) however, the improvement dimishes at $d = 8$ and beyond, due to the increasing prevalence of duplicates and hash collisions. These trends hold across both the ISP and data center scenarios, for a variety of measured heavy hitters k and memory sizes. Hence, we choose $d = 6$ table stages for all further experiments with HashPipe.

5.2 Accuracy of HashPipe

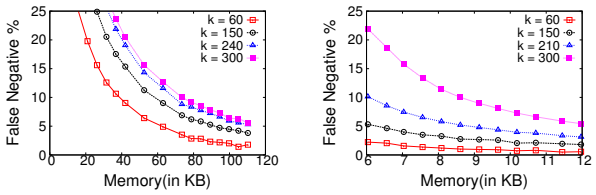
We plot error *vs.* memory tradeoff curves for HashPipe, run with $d = 6$ table stages. Henceforth, unless mentioned otherwise, we run the data center trace at the single link packet rate.

False negatives. Fig. 5 shows the false negatives as memory increases, with curves corresponding to different numbers of reported heavy hitters k. We find that error decreases with allocated memory across a range of k values, and settles to under 10% in all cases on the ISP trace at 80KB of memory, which corresponds to 4500 counters. In any one trial with the ISP trace, there are on average 400,000 flows, which is two orders of magnitude higher than the number of counters we use. In the data center trace, the error settles to under 10% at just 9KB of memory (520 counters) for all the k values we tested.[3]

These results also enable us to understand the interplay between k and the memory size required for a specific accuracy. For a 5% false negative rate in the ISP trace, the memory required for $k = 60$ is 60KB ($3375 \approx 55k$ counters), whereas the memory required for $k = 300$ is 110KB ($6200 \approx 20k$ counters). In general, the factor of k required in the number of counters to achieve a particular accuracy reduces as k increases.

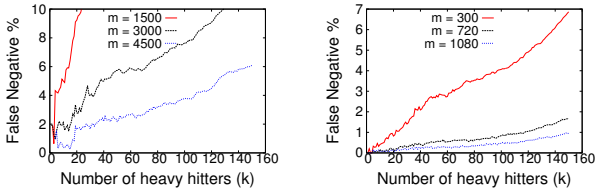
Which flows are likely to be missed? It is natural to ask which flows are more likely missed by HashPipe. Fig. 6 shows how the false negative rate changes as the number of desired heavy hitters k is varied, for three different total memory sizes. We find that *the heaviest flows are least likely to be missed, e.g.,* with false negatives in the 1-2% range for the top 20 flows, when using 3000 counters in the ISP trace. This trend is intuitive, since HashPipe prioritizes the retention of larger flows in the table (§3.1). Most of the higher values of false negative errors are at larger values of k, meaning that the smaller

[3]The minimum memory required at $k = 300$ is 6KB (≈ 300 counters).



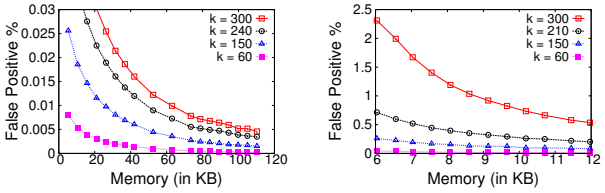
(a) ISP backbone (b) Data center

Figure 5: False negatives of HashPipe with increasing memory. Each trial trace from the ISP backbone contains an average of 400,000 flows, yet HashPipe achieves 5-10% false negatives for the top 60-300 heavy hitters with just 4500 flow counters.



(a) ISP backbone (b) Data center

Figure 6: False negatives against different numbers of reported heavy flows k. The heavier a flow is, the less likely that it is missed.



(a) ISP backbone (b) Data center

Figure 7: HashPipe has false positive rates of 0.01%-0.1% across a range of table sizes and reported heavy hitters in the ISP trace, and under 3% in the data center trace.

of the top k flows are more likely to be missed among the reported flows.

False positives. Fig. 7 shows the false positives of HashPipe against varying memory sizes, exhibiting the natural trend of decreasing error with increasing memory. In particular, we find that with the ISP trace, false positive rates are very low, partly owing to the large number of small flows. On all curves, the false positive rate is smaller than 0.1%, dropping to lower than 0.01% at a table size of 80KB. In the data center trace, we find that the false positive rate hovers under 3% over a range of memory sizes and heavy hitters k.

In summary, HashPipe performs well both with the ISP backbone and data center traces, recognizing heavy-hitter flows in a timely manner, *i.e.,* within 20 seconds and 1 second respectively, directly in the data plane.

5.3 HashPipe *vs.* Existing Solutions

Comparison baselines. We compare HashPipe against two schemes—representative of sampling and sketching—which are able to estimate counts in the switch directly (§2.2). We use the same total memory as earlier, and measure the top $k = 150$ flows. We use the ISP backbone trace henceforth (unless mentioned otherwise), and compare HashPipe with the following baseline schemes.

(1) Sample and Hold: We simulate sample and hold [17] with a flow table that is implemented as a *list*. As described in §2.2, we liberally allow incoming packets to look up flow entries anywhere in the list, and add new entries up to the total memory size. The sampling probability is set according to the available flow table size, following recommendations from [17].

(2) Count-min sketch augmented with a 'heavy flow' cache: We simulate the count-min sketch [14], but use a flow cache to keep flow keys and exact counts starting from the packet where a flow is *estimated to be heavy* from the sketch.[4] We liberally allow the count-min sketch to use the offline-estimated exact count of the kth heaviest flow in the trace, as a threshold to identify heavy flows. The flow cache is implemented as a hash table that retains only the pre-existing flow key on a hash collision. We allocate half the available memory each to the sketch and the flow cache.[5]

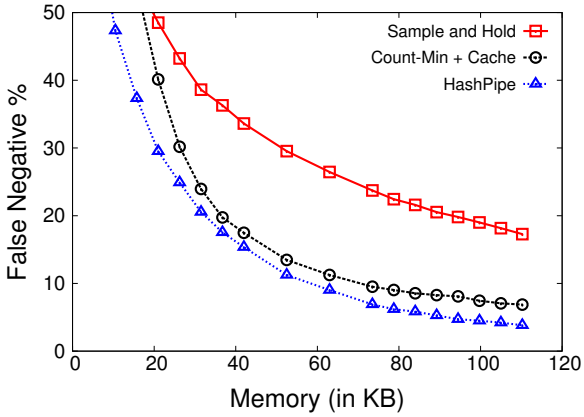


Figure 8: False negatives of HashPipe and other baselines. HashPipe outperforms sample and hold and count-min sketch over the entire memory range.

[4] A simpler alternative—mirroring packets with high size estimates from the count-min sketch—requires collecting ≈ 40% of the packets in the data center trace played at the link rate.

[5] We follow guidance from [17, page 288] to split the memory.

False negatives. Fig. 8 shows false negatives against varying memory sizes, for the three schemes compared. We see that HashPipe outperforms sample and hold as well as the augmented count-min sketch over the entire memory range. (All schemes have smaller errors as the memory increases.) Notably, at 100KB memory, HashPipe has 15% smaller false negative rate than sample and hold. The count-min sketch tracks the error rate of HashPipe more closely from above, staying within a 3-4% error difference. Next, we understand where the errors occur in the baseline schemes.

Where are the errors in the other baselines? Fig. 9 shows the count estimation error (%) averaged across flows whose true counts are higher than the x-value, when running all the schemes with 26KB memory.

Sample and hold can make two kinds of estimation errors. It can miss the first set of packets from a heavy flow because of not sampling the flow early enough, or (less likely) miss the flow entirely due to not sampling or the flow table becoming full. Fig. 9 shows that sample and hold makes the former kind of error even for flows with very high true counts. For example, there are relative errors of about 10% even for flows of size more than 80,000 packets. As Fig. 8 shows, the errors becomes less prominent as the memory size increases, since the sampling rate increases too.

The count-min sketch makes errors because of its inability to discriminate between heavy and light flows during hash collisions in the sketch. This means that a *light* flow colliding with heavy flows may occupy a flow cache entry, preventing a heavier flow later on from entering the flow cache. For instance in Fig. 9, even flows as large as 50,000 packets can have estimation errors close to 20%. However, as Fig. 8 shows, the effect of hash collisions becomes less significant as memory size increases.

On the other hand, HashPipe's average error on flows larger than 20,000 packets—which is 0.2% of the total packets in the interval—is negligible (Fig. 9). HashPipe has 100% accuracy in estimating the count of flows larger than 30,000 packets.

5.4 HashPipe *vs.* Idealized Schemes

We now compare HashPipe against the idealized algorithms it is derived from (§3), namely space saving [28] and HashParallel.

There are two reasons why HashPipe may do badly relative to space saving: (i) it may evict a key whose count is much higher than the table minimum, hence missing heavy items from the table, and (ii) it may allow too many duplicate flow keys in the table (§3.4), reducing the memory available for heavy flows, and evict heavy flows whose counts are underestimated due to the duplicates. We showed that duplicate keys are not very prevalent in §5.1; in what follows, we understand their effects on false negatives in the reported flows.

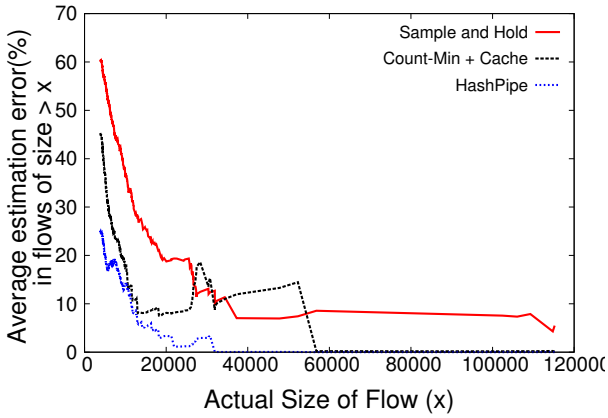


Figure 9: Comparison of average estimation error (%) of flows whose true counts are higher than the x-value in the graph. Sample and hold underestimates heavy flows due to sampling, and count-min sketch misses heavy flows due to lighter flows colliding in the flow cache. HashPipe has no estimation errors for flows larger than 30,000 packets.

How far is the subsampled minimum from the true table minimum? Fig. 10 shows the complementary CDF of the minimum count that was chosen by HashPipe, obtained by sampling the algorithm's choice of minimum at every 100th packet in a 10 million packet trace. We don't show the corresponding CCDF for the absolute minimum in the table, which only takes on two values—0 and 1—with the minimum being 1 more than 99% likely.[6] We see that the chance that HashPipe chooses a certain minimum value decreases rapidly as that value grows, judging from the straight-line plot on log scales in Fig. 10. For example, the minimum counter has a value higher than 5 less than 5% of the time. There are a few larger minimum values (*e.g.,* 100), but they are rarer (less than 0.01% of the time). This suggests that it is unlikely that HashPipe experiences large errors due to the choice of a larger minimum from the table, when a smaller counter exists.

Comparison against space saving and HashParallel. We compare the false negatives of the idealized schemes and HashPipe against varying memory sizes in Fig. 11 and Fig. 12, when reporting two different number of heavy hitters, k=150 and k=60 respectively. Two features stand out from the graph for both k values. First, wherever the schemes operate with low false negative error (say less than 20%), the performance of the three schemes is comparable (*i.e.,* within 2-3% of each

[6]Note that this is different from the minimum of space saving, whose distribution contains much larger values.

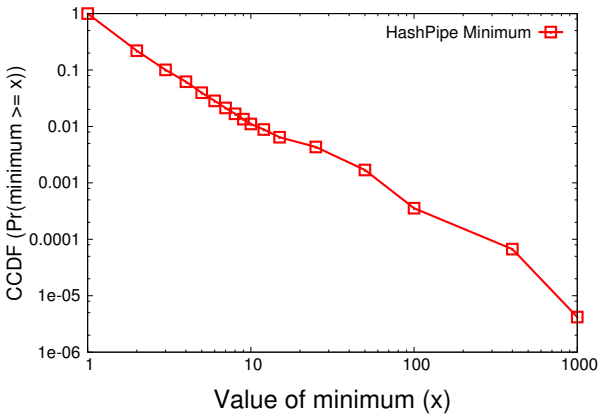


Figure 10: Complementary CDF of the minimum chosen by HashPipe on $26KB$ of memory, sampled every 100th packet from a trace of 10 million packets.

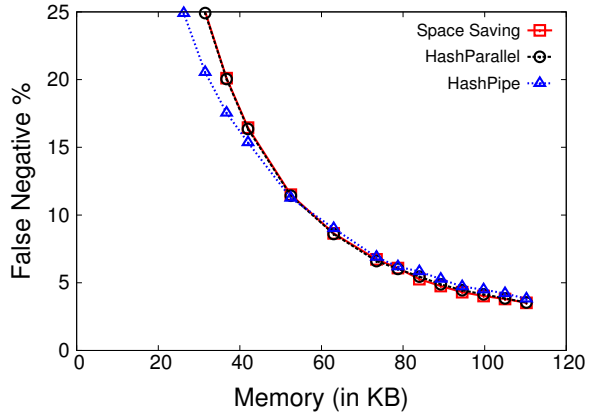


Figure 11: Comparison of false negatives of HashPipe to idealized schemes when detecting k=150 heavy hitters. In some memory regimes, HashPipe may even outperform space saving, while HashParallel closely tracks space saving.

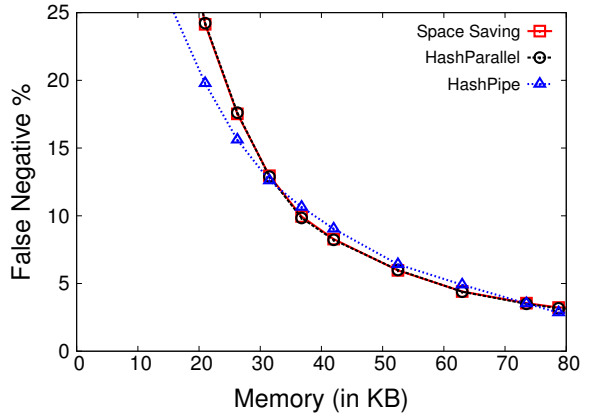


Figure 12: Comparison of false negatives of HashPipe to idealized schemes when detecting k=60 heavy hitters.

other). Second, there are small values of memory where HashPipe *outperforms* space saving.

Why is HashPipe outperforming space saving? Space saving only guarantees that the kth heaviest item is in the table when the kth item is larger than the average table count (§3.1). In our trace, the 60th item and 150th item contribute to roughly 6000 and 4000 packets out of 10 million (resp.), which means that they require at least[7] 1700 counters and 2500 counters in the table (resp.). These correspond to memory sizes of 30KB and 45KB (resp.). At those values of memory, we see that space saving starts outperforming HashPipe on false negatives.

We also show why space saving fails to capture the heavier flows when it is allocated a number of counters smaller than the minimum number of counters mentioned above. Note that HashPipe attributes every packet to its flow entry correctly (but may miss some packets entirely), since it always starts a new flow at counter value 1. However, space saving increments *some counter* for every incoming packet (Algorithm 1). In contexts where the number of active flows is much larger than the number of available counters (*e.g.*, 400,000 flows with 1200 counters), this can lead to some flows having enormously large (and grossly overestimated) counters. In contrast, HashPipe keeps the counter values small for small flows by evicting the flows (and counts) entirely from the table.

At memory sizes smaller than the thresholds mentioned above, incrementing a counter for each packet may result in *several small flows* catching up to a heavy flow, leading to significant false positives, and higher likelihood of evicting truly heavy flows from the table. We show this effect on space saving in Fig. 13 for k=150 and m=1200 counters, where in fact $m = 2500$ counters are required as described earlier. The distribution of the number of keys contributing to a false positive flow counter in the table is clearly shifted to the right relative to the corresponding distribution for a true positive.

Impact of duplicate keys in the table. Finally, we investigate how duplicate keys and consequent underestimation of flow counts in the table may affect the errors of HashPipe. In Fig. 14, we show the benefits of reporting *more than k counters* on false negatives, when the top k=300 flows are requested with a memory size of m=2400 counters. While the false negative rates of space saving and HashParallel improve significantly with overreporting, the errors for HashPipe remains flat throughout the interval, dropping only around 1800 reported flows. We infer that most heavy flows are retained somewhere in the table for space saving and HashParallel, while HashPipe underestimates keys sufficiently often that

[7] 10 million / 6000 ≈ 1700; 10 million / 4000 ≈ 2500.

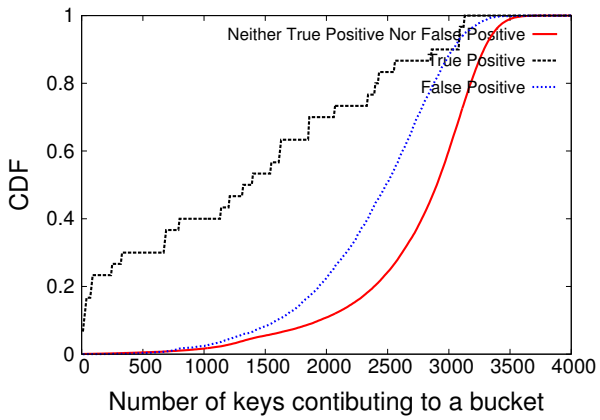


Figure 13: CDF of the number of distinct flows contributing to a counter in space saving's table, when identifying k=150 heavy flows with m=1200 counters. We show three distributions according to the flow's label after the experiment.

they are completely evicted—to the point where overreporting does not lower the false negative errors. We find that overreporting flows only increases the false positive errors slightly for all schemes, with values staying between 0.1-0.5% throughout.

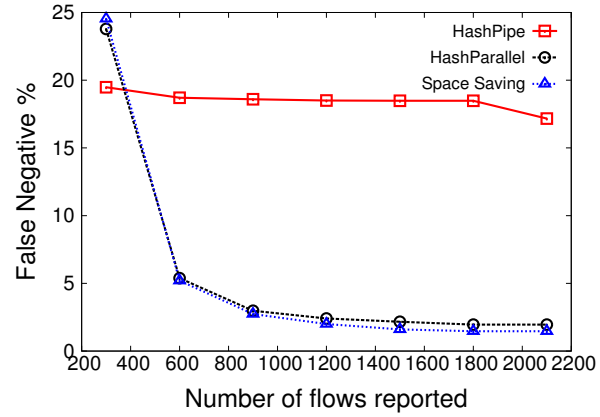


Figure 14: Benefits of overreporting flows on false negative errors with k=300 heavy flows and m=2400 counters. While space saving and HashParallel improve significantly by reporting even $2k$ flows, HashPipe does not, because of evictions due to duplicate entries.

6 RELATED WORK

Applications that use heavy hitters. Several applications use information about heavy flows to do better traffic management or monitoring. DevoFlow [16] and Planck [35] propose exposing heavy flows with low overhead as ways to provide better visibility and reduce congestion in the network. UnivMon [25] uses a top-k detection sketch internally as a subroutine in its "universal" sketch, to determine more general statistics about the network traffic. There is even a P4 tutorial application on switch programming, that performs heavy hitter detection using a count-min-sketch-like algorithm [11].

Measuring per-flow counters. Prior works such as FlowRadar [24] and CounterBraids [26] have proposed schemes to measure accurate per-flow traffic counters. Along similar lines, hashing schemes like cuckoo hashing [33] and d-left hashing [43] can keep per-flow state memory-efficiently, while providing fast lookups on the state. Our goal is not to measure or keep all flows; just the heavy ones. We show (§5) that HashPipe uses 2-3 orders of magnitude smaller memory relative to having per-flow state for all active flows, while catching more than 90% of the heavy flows.

Other heavy-hitter detection approaches. The multi-resolution tiling technique in ProgME [46], and the hierarchical heavy-hitter algorithm of Jose *et al.* [21] solve a similar problem as ours. They estimate heavy hitters in traffic *online*, by iteratively "zooming in" to the portions of traffic which are more likely to contain heavy flows. However, these algorithms involve the control plane in running their flow-space-partitioning algorithms, while HashPipe works completely within the switch pipeline. Further, both prior approaches require temporal stability of the heavy-hitter flows to detect them over multiple intervals; HashPipe determines heavy hitters using counters maintained in the same interval.

7 CONCLUSION

In this paper, we proposed an algorithm to detect heavy traffic flows within the constraints of emerging programmable switches, and making this information available within the switch itself, as packets are processed. Our solution, HashPipe, uses a pipeline of hash tables to track heavy flows preferentially, by evicting lighter flows from switch memory over time. We prototype HashPipe with P4, walking through the switch programming features used to implement our algorithm. Through simulations on a real traffic trace, we showed that HashPipe achieves high accuracy in finding heavy flows within the memory constraints of switches today.

ACKNOWLEDGMENTS

We thank the anonymous SOSR reviewers, Anirudh Sivaraman, and Bharath Balasubramanian for their feedback. This work was supported partly by NSF grant CCF-1535948, DARPA grant HR0011-15-2-0047, and gifts from Intel and Huawei. We also thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.

[2] Arista Networks. Arista 7050x switch architecture, 2014. https://www.corporatearmor.com/documents/Arista_7050X_Switch_Architecture_Datasheet.pdf.

[3] Barefoot Networks. Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, 2011.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.

[8] CAIDA. The CAIDA UCSD Anonymized Internet Traces 2016 - March. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[9] Cavium. Cavium and XPliant Introduce a Fully Programmable Switch Silicon Family Scaling to 3.2 Terabits per Second. http://cavium.com/newsevents- Cavium-and-XPliant-Introduce-a-Fully-Programmable-Switch-Silicon-Family.html.

[10] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Springer ICALP*, 2002.

[11] S. Choi. Implementing Heavy-Hitter Detection, 2016. https://github.com/p4lang/tutorials/tree/master/SIGCOMM_2016/heavy_hitter.

[12] Cisco Networks. Netflow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[13] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *VLDB Endowment*, 2008.

[14] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[15] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1), 2005.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM*, 2011.

[17] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM Trans. Computer Systems*, 21(3), 2003.

[18] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. In *ACM SIGCOMM*, 2000.

[19] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM*, 2014.

[20] John Wawrzynek and Krste Asanovic and John Lazzaro and Yunsup Lee. Memory (lecture), 2010. https://inst.eecs.berkeley.edu/~cs250/fa10/lectures/lec08.pdf.

[21] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *USENIX Hot-ICE*, 2011.

[22] M. Kumar and K. Prasad. Auto-learning of MAC addresses and lexicographic lookup of hardware database. US Patent App. 10/747,332.

[23] A. Lakhina, M. Crovella, and C. Diot. Characterization of network-wide anomalies in traffic flows. In *ACM IMC*, 2004.

[24] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, 2016.

[25] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[26] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: A novel counter architecture for per-flow measurement. In *ACM SIGMETRICS*, 2008.

[27] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB Endowment*, 2002.

[28] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 2005.

[29] J. Misra and D. Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[30] NANOG mailing list. SFlow vs NetFlow/IPFIX. https://mailman.nanog.org/pipermail/nanog/2016-February/thread.html#84418.

[31] R. Ozdag. Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[32] P4 Language Consortium. P4 Switch Behavioral Model. https://github.com/p4lang/behavioral-model.

[33] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[34] P. Phaal. SFlow sampling rates. http://blog.sflow.com/2009/06/sampling-rates.html.

[35] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM*, 2014.

[36] O. Rottenstreich and J. Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Trans. Netw.*, 2017.

[37] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *ACM IMC*, 2004.

[38] SFlow. http://sflow.org/.

[39] Simon Horman. Packet recirculation, 2013. https://lwn.net/Articles/546476/.

[40] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, 2016.

[41] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM*, 2016.

[42] The P4 Language Consortium. The P4 Language Specification, Version 1.1.0 - Release Candidate, January 2016. http://p4.org/wp-content/uploads/2016/03/p4_v1.1.pdf.

[43] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[44] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2010.

[45] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

[46] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards programmable network measurement. In *ACM SIGCOMM*, 2007.