SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches

Hun Namkung[⋆], Zaoxing Liu[†], Daehyeok Kim[⋆§], Vyas Sekar[⋆], Peter Steenkiste[⋆]
[⋆]*Carnegie Mellon University,* [†]*Boston University,* [§]*Microsoft*

Abstract

Sketching algorithms or sketches enable accurate network measurement results with low resource footprints. While emerging programmable switches are an attractive target to get these benefits, current implementations of sketches are either inefficient and/or infeasible on hardware. Our contributions in the paper are: (1) systematically analyzing the resource bottlenecks of existing sketch implementations in hardware; (2) identifying practical and correct-by-construction optimization techniques to tackle the identified bottlenecks; and (3) designing an easy-to-use library called *SketchLib* to help developers efficiently implement their sketch algorithms in switch hardware to benefit from these resource optimizations. Our evaluation on state-of-the-art sketches demonstrates that SketchLib reduces the hardware resource footprint up to 96% without impacting fidelity.

1 Introduction

The ability to monitor network traffic is necessary for various network management tasks such as traffic engineering, anomaly detection, load balancing, and resource provisioning [10, 13, 27, 29, 43, 45, 54]. In this respect, recent developments in programmable switches and attendant languages [9, 14] make it possible to support richer fine-grained and real-time monitoring capabilities.

With this network programmability, sketch-based monitoring has emerged as a promising alternative to traditional sampling-based techniques [19, 49]. At a high-level, sketch algorithms consist of updating multiple counter arrays with a series of independent *hash function calls* and *counter updates*. Sketch-based approaches have been developed to support a broad spectrum of measurement tasks with provable resource-accuracy trade-offs, including heavy-hitter detection or quantile estimation (e.g., [17, 21]), general estimation capabilities (e.g., UnivMon [41]), and more expressive multidimensional analytics (e.g., R-HHH [12]).

While prior efforts have demonstrated the feasibility of expressing sketches using these language APIs [32, 41, 46, 53], implementing sketches efficiently in hardware remains an open challenge. For example, off-the-shelf sketch implementations often cannot run with the desired accuracy levels due to insufficient hardware resources (see §3). Indeed, some proposed sketches (e.g., [41]) are infeasible as implemented, or even if they are feasible, consume significant resources.

Even if more hardware resources may become available, so too do operators' demands of in-switch applications, and the resources consumed by sketches will be unavailable for other switch functions. Thus, it is essential to explore if, and how, we can efficiently realize sketch-based telemetry on programmable switches. This is the central question that this paper tackles. Specifically, we focus on programmable hardware switches based on the Reconfigurable Match-Action Tables (RMT) paradigm [1].

We identify and analyze four key resource bottlenecks for realizing sketches on RMT switch hardware:

- *Hash calls:* Sketches make a number of counter updates based on independent hash functions, requiring a large number of hash calls in hardware.
- *Memory accesses:* Sketches need to access on-chip memory (e.g., SRAM) for counter updates, but the number of memory accesses per packet is limited in hardware.
- *Pipeline stages:* Some sketches need to select a subset of counter arrays for counter updates [23, 37, 41]. However, implementing this naively can cause a long chain of sequential computation dependencies which stresses the limited number of switch pipeline stages.
- *Resources for tracking heavy flowkeys:* Some sketches need to keep track of the flowkeys identifying the heavy hitters (e.g., 5-tuple, source IP, or destination IP) [12, 17, 21, 36, 41]. Common structures such as priority queues or heaps used in software are not supported on programmable switches and existing solutions entail undesirable tradeoffs between miss rate, data plane memory, and control plane bandwidth.

Having identified these bottlenecks, our contribution is a careful synthesis of known and novel optimizations into a practical library for enabling efficient sketch implementations atop the RMT architecture. While some of these build on prior work in optimizing sketching for other targets such as software switches, FPGAs, and embedded platforms [40, 51, 52, 55], our main contribution is in realizing feasible and effective optimizations based on our bottleneck analysis and translating them into the switch hardware setting. For example, to reduce the number of hash calls, we identify opportunities to consolidate and reuse hash results across multiple counter updates [24, 35]. Similarly, we identify an opportunity to reduce the pipeline stages by eliminating code dependencies based on longest prefix matching using TCAM [55]. We reduce the memory accesses by refactoring sketch algorithms and removing unnecessary memory accesses. We also develop practical flowkey tracking mechanisms that are feasible in hardware. Note that all optimizations preserve correctness while reducing the resource footprint.

To make it easy for sketch developers to benefit from these optimizations with minimal effort, we implement *SketchLib*, an easy-to-use API using the P4 language [14]. These optimizations can be applied to a broad spectrum of classical sketches (e.g., [17, 21, 36]) and recent innovations (e.g., [12, 41]). We qualitatively evaluate the suitability of SketchLib for 19 published sketches and observe that 15 of them can be expressed and can benefit from one or more of our optimizations. We acknowledge that not all optimizations are applicable for every sketch and we envision sketch developers using our API to adopt the relevant optimizations.

We quantitatively evaluate the utility of SketchLib in improving 7 of the 15 applicable sketches covering a diverse set of target telemetry tasks: Count Sketch (CS) [17], PCSA [25], MRAC [37], Multi-resolution Bitmap [23], Hierarchical Heavy Hitters [12], and UnivMon [41]. Our evaluation using a range of packet traces empirically confirms that our optimizations provide similar accuracy ($\leq 1.9\%$) with substantially (up to 96%) reduced resource usage. Furthermore, some complex sketches (e.g., UnivMon) that were previously infeasible on current hardware become feasible.

Contributions and Roadmap. To summarize, we make the following contributions:

- **Bottleneck Analysis** (§3): We identified four key resource bottlenecks for sketch implementations on the hardware programmable switch.
- **Optimizations** (§4): We identify and synthesize practical correctness-preserving optimizations to address the bottlenecks for sketches on switch hardware.
- **API Implementation** (§5): We design a convenient API to make our optimizations easy to use for developers who implement sketches on RMT programmable switches.[1] We verified significant resource benefits on a broad range of sketching algorithms.

[1] SketchLib is publicly available at https://github.com/SketchLib.

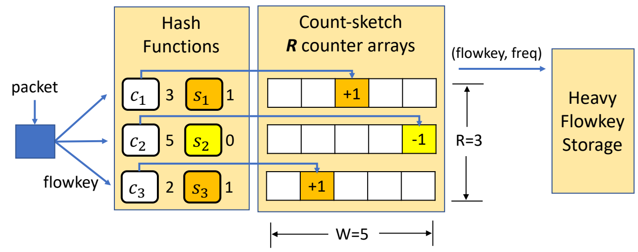


Figure 1: Count Sketch has three components - hash computations, multiple counter arrays, and heavy flowkey storage.

```
control ingress // R-HHH
{
  V = randomInt(1, L);

  if (V == 1) {
    key = srcIP/32;
    apply(CS_level_1,key);
  }
  if (V == 2) {
    key = srcIP/24;
    apply(CS_level_2,key);
  }
  if (V == 3) {
    key = srcIP/16;
    apply(CS_level_3,key);
  }
  ...
}
```

(a) R-HHH

```
control ingress // UnivMon
{
  key = srcIP/32;

  apply(CS_level_1, key);
  apply(compute_hash_h1, key);

  if (h1 == 1) {  // 0 or 1
    apply(CS_level_2, key);
    apply(compute_hash_h2, key);

    if (h2 == 1) {
      apply(CS_level_3, key);
      apply(compute_hash_h3, key);

      if (h3 == 1) {
        ...
}
```

(b) UnivMon

Figure 2: Simplified P4 code of existing multi-level sketches.

2 Background

In this section, we start by providing some background on sketching algorithms and programmable switch architecture. We then describe how the sketch code is mapped onto the hardware resources.

2.1 Background on Sketches

Sketching algorithms or *sketches* are randomized approximation algorithms that are designed to compute different observed statistics on a given data stream during every measurement time interval called *epoch*. In network monitoring, prior work has shown that sketches (e.g., [12, 17, 21, 32, 40–42, 46, 53]) offer better resource-accuracy tradeoffs relative to traditional techniques that rely on sampling (e.g., NetFlow [19]). Our focus in this paper is not to develop new sketches but to enable efficient sketch realizations on programmable switches. To better understand the different resource requirements of sketches, we classify prior sketching work into two categories:

1. Single-level sketches: As a canonical example, consider the *count sketch* (CS) [17] for heavy hitter detection shown in Fig. 1. A single-level sketch such as Count Sketch maintains a 2D-array of counters: R independent counter arrays with size of W; i.e., $R \times W$ memory counters. As each packet arrives, we extract a flowkey from the packet (e.g., srcIP, IP 5-tuple). On this key, we compute two independent hash functions c_i and s_i, corresponding for each row i. c_i is used to select a specific column and s_i is a 1-bit hash used to determine either to increase or decrease the counter.

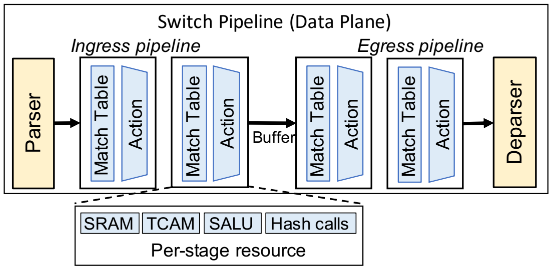


Figure 3: RMT switch architecture.

The total number of hash computations is $2R$. Count Sketch requires additional memory space for storing heavy flowkeys whose estimated flow counts are above a threshold. Other single-level sketches requiring a 2D-array include the count-min sketch (CMS) [21], k-ary (Kary) sketch [36]. Some single-level sketches like HyperLogLog (HLL) [26] only need a 1D array data structure.

2. Multi-level sketches: Conceptually, these consist of multiple single-level sketches to enable richer queries. For instance, R-HHH and UnivMon can use multiple count sketches, called levels (e.g., L levels of $R \times W$ counters). R-HHH supports detection of hierarchical heavy hitters, which detects heavy hitters based on different lengths of IP prefixes and UnivMon provides more general estimation capabilities. Other sketches like PCSA, MRAC, and multi resolution bitmap (MRB) [23,25,37] use multiple 1D-array single-level sketches. Multi-level sketches typically select a subset of counter arrays to issue counter updates for a given flowkey. For instance, as shown in Fig. 2a, R-HHH randomly selects one level of count sketch using a level-specific key (e.g., IP prefix) to update per packet. In contrast, UnivMon uses an additional sampling stage using hash functions that return 0 or 1 to select levels for update, as shown in Fig. 2b.

2.2 Programmable Switch Hardware

Our focus in this paper is programmable switch hardware based on the Reconfigurable Match-Action Tables (RMT) paradigm [15]. A canonical commercial realization of this architecture is the Intel Tofino switch chip [1]. Based on public documentation and conversations with vendors, we believe that while other programmable switches (e.g., Broadcom Trident [2]) may have different hardware resource allocation strategies, the architectural bottlenecks for sketches are likely similar. We leave it as future work to extend SketchLib to other hardware targets.

Hardware architecture. RMT-based programmable switches have a pipeline of reconfigurable match-action tables in the data plane, as shown in Fig. 3. There are constraints in packet processing pipeline to meet the line-rate processing requirement. For example, at each stage, a packet can access a limited amount of compute and memory resources. Each stage has an identical design with the same types of resources. To provide flexible match-action operations, each stage has a *match table* that matches packet

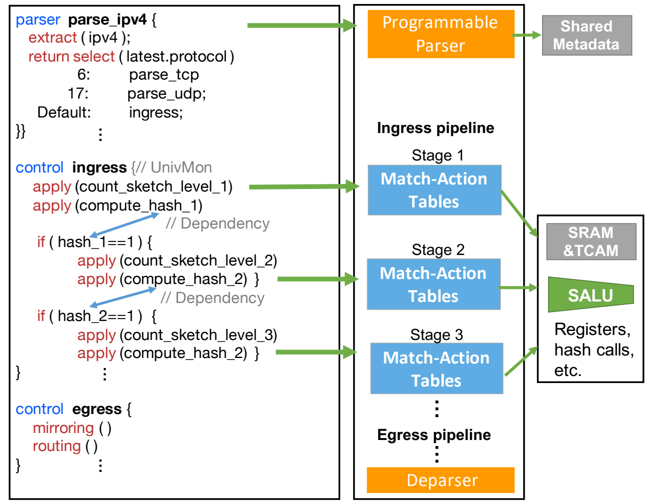


Figure 4: Mapping P4 code to switch resources.

headers to specific values followed by an action unit that executes a set of simple instructions, depending on the output of the matching unit.

Key hardware resources. We now briefly describe the key hardware resources available in each pipeline stage. First, there are a number of hardware *hash function calls* (hash calls) per pipeline stage. They are used to compute hash functions (e.g., CRC with user-defined polynomials) over packet header fields or metadata to support operations such as load balancing and table lookups. Each pipeline stage also has a fixed amount of SRAM that can be used to maintain state, for example counter arrays. *Stateful ALUs* (SALUs) are hardware resources that allow one read and one write operation to the stateful object in SRAM. Each SALU can be used for counter update operations such as counter increment or decrement. Finally, each pipeline stage is also equipped with some amount of *ternary content-addressable memory* (TCAM) that can be used for wildcard matches over header fields. Overall, the amount of these resources is fixed at hardware design time, and it is limited. For example, a commercial programmable switch today is equipped with (at most) 10 SALUs, 10 hash calls, 10 MBs of SRAM and TCAM per pipeline stage with a total of 12 pipeline stages [15, 43, 56].[2]

The data plane can interact with the switch control plane for additional processing. However, the switch control plane is not designed for real-time processing, e.g., the bandwidth to the control plane is limited and the response time is high. So it is only useful for infrequent operations.

2.3 P4 Programming and Compilation

Programs for RMT switches are written in the P4 language [14] as illustrated in Fig. 4. At a high-level, a P4 program consists of the following components. First, a packet parser parses the header fields of each packet and stores the extracted fields into metadata. Second, a series of match-action

[2]The other absolute resource numbers are proprietary.

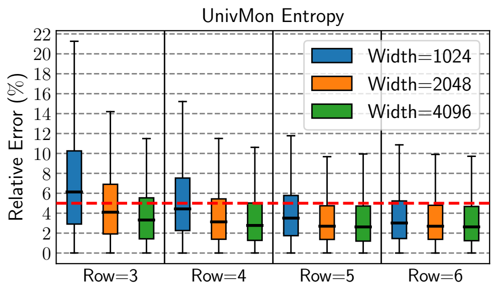


Figure 5: UnivMon entropy estimation error for different configurations. Dotted red line indicates target accuracy.

operations are executed based on the *match-action* abstraction, e.g., matching a specific header field and update a register as an action. The action is specified by special functions that map operations to hardware resources, e.g., functions for hashing and accessing memory. Finally, the P4 program defines the packet forwarding behaviors, e.g., routing a packet to an egress port, recirculating it in the pipeline, or forwarding it to the switch control plane.

The P4 compiler maps the P4 program into a static pipeline realization. The compiler analyzes the *dependencies* between operations in the P4 program, to map the program on to the pipeline stages. For instance, given the code snippet in Fig. 4, the resolution of each `if`-clause depends on the previous hash result. Because of this dependency, two consecutive `if`-clauses cannot run in parallel, so the compiler has to map them to different pipeline stages in order for them to run sequentially. If a mapping of a whole program is possible considering hardware constraints, packets are guaranteed to be processed at line rate; otherwise compilation fails. Note that vendor-specific compiler backends are typically proprietary.

3 Bottleneck Analysis

In this section, we consider three exemplar sketches (single-level: count sketch; multi-level: R-HHH and UnivMon) to quantify the resource bottlenecks. We implement them in P4 based on the logic described in prior work [17, 23, 25, 26, 37, 41] similar to the structure presented in Fig. 2.

3.1 Methodology and Setup

Configuring sketches. Running sketches entails picking parameters (e.g., the count (R) and size (W) of counter arrays) to trade-off the accuracy vs. resource use. We envision an operator configuring the sketches with some target accuracy goal, e.g., the median error should be less than 5%. Operators can use trace-driven analysis to pick reasonable operating regimes for these parameters.

As an example, Fig. 5 illustrates this trade-off for entropy estimation using UnivMon. The figure shows the estimation accuracy using an hour-long inter-ISP packet trace captured on a OC-192 link [7] with different parameters R and W for count sketches and $L = 16$ levels. We see that the error decreases as we increase the number of rows (R) and width (W) for count sketches. Naturally, the higher accuracy configurations incur more hardware resources. For our bottleneck analysis, we target an accuracy of under 5% median error (dotted red line in Fig. 5), which we achieve with minimal resource use with the configuration $R = 3$ and $W = 2048$. We repeat the analysis for count sketch and R-HHH and consider a similar operating regime for these sketches as well.

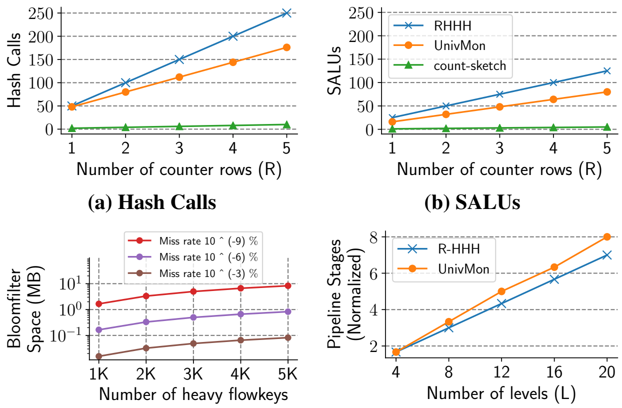
Estimating resource footprint. For a given set of sketch parameters, the most direct way to measure the required hardware resources is to compile the code and run it on the hardware. However, this limits our analysis to currently available platforms. In order to support "what if" analysis for hardware with different resources (e.g., more pipeline stages), we extended an existing open source tool for mapping P4 programs to the RMT hardware, which we will refer to as RMT resource mapper [34]. Specifically, we address three issues to extend RMT resource mapper for our analysis:

- *Inputs:* The input to Tofino compiler is P4-16 code with some hardware-specific primitives whereas RMT resource mapper accepts only P4-14 code [8]. Thus, we first convert our P4-16 code into equivalent P4-14 code. Then, we convert Tofino-specific primitives to equivalent ones specified in the language specification. For instance, we replace Tofino-specific primitives for accessing registers with *register_read* and *register_write*.
- *Resources:* First, RMT resource mapper does not model hash calls and SALUs in their original design. Thus, we extend RMT resource mapper to model hash calls and SALUs and added the corresponding optimization constraints for assignment of these new resources. Second, we observed that RMT resource mapper assigns memory even for tables without any entries and action data. To fix this disconnect, we decouple the memory/table assignment.
- *Objective:* RMT resource mapper supports different optimization objectives: minimizing latency, power, or pipeline stages. The objective of minimizing pipeline stages is the most suitable because it gives resource mappings that are closest to those generated by the Tofino compiler.

With these fixes in place, we validate our extensions by comparing the resource usages between RMT resource mapper and Tofino compiler for a wide range of sketches and configurations, for the cases that are feasible on current hardware. Based on the measurement results, we conclude that our modified RMT resource mapper is a good proxy of Tofino compiler as it captures the relevant resource constraints, and its resource allocation results are close to that of Tofino compiler (see Appendix A for more details).

3.2 Identified Bottlenecks

Using the RMT resource mapper, we measure the usage of each type of resources based on the output of the compiler for three sketches: Count Sketch, R-HHH, and UnivMon. For the purpose of bottleneck analysis, we use a base configuration



(a) Hash Calls **(b) SALUs** **(c) Flowkey Space** **(d) Pipeline Stages**

Figure 6: Resource bottlenecks for sketch implementations.

of: $W = 2048$, $R = 3$, and $L = 16$ for UnivMon and $L = 25$ for R-HHH [12], which provides an error of up to 15% when processing packets from an inter-ISP packet trace [7]. We choose the value for L from the original papers [12, 41].

Fig. 6 illustrates how the use of four bottleneck resources depends on key sketch parameters. While the amount of available hardware resources can differ across hardware vendors and versions, we see that resource usage increases rapidly as we need more counters to meet higher accuracy requirements. While we cannot report exact resource usages due to proprietary reasons, we note that UnivMon and R-HHH are infeasible today on the hardware for many configurations. Perhaps more importantly, switches must also support tasks other than sketch-based telemetry (e.g., [33, 43]). Thus, it is critical to reduce the resource footprint of the sketches to ensure they can co-exist with other switch functions.

B1. Hash calls: Recall that count sketch needs $2R$ hash calls per packet (§2.1), matching the results in Fig. 6a. UnivMon and R-HHH execute one count sketch per level L. As a result, R-HHH needs $L \cdot 2R$ hash calls. UnivMon needs to compute an additional L 1-bit hash calls in its sampling stage, adding up to $L \cdot (2R + 1)$ hash calls.

At first glance, it may seem that the number of hash calls is not a bottleneck as these are called on demand per packet. While this is true in a software setting, where only the required calls are performed on demand, hashing on hardware is different. On a hardware switch, all hash calls appearing in the code need to be *pre-allocated* since execution at line rate must be guaranteed for all possible execution paths. This increases resource requirements, even if hash calls need not be executed. For example, even though UnivMon and R-HHH (Fig. 2) may not update all levels of count sketches for all packets, all hash resources must be pre-allocated.

B2. Memory accesses: Count Sketch maintains R counter arrays (§2.1) and for each row it must read one counter from memory and update its value. This means that count sketch needs R counter updates per packet, requiring R Stateful ALUs (SALUs) as shown in Fig. 6b. When the compiler compiles the P4 code of UnivMon and R-HHH in Fig. 2, it allocates separate memory regions and SALUs for each level of count sketches, thus SALU requirements are proportional to the number of levels L. Since we need R memory processes per packet for the count sketch at each level, we need a total $L{\cdot}R$ SALUs for R-HHH and UnivMon. This makes memory access hardware (SALU) a bottleneck (Fig. 6b). Similar to hash resources, SALUs need to be *pre-allocated* at compile time, even if they may remain unused.

B3. Resources for tracking heavy flowkeys: Many sketches need to track heavy flowkeys to enable downstream analytics tasks. Typically, these sketches store heavy flowkeys in a separate data structure (e.g., heap or priority queue) [17, 41].

In practice, however, the exact details of if/how this can be realized on switch hardware are unclear. Specifically, a heap or priority queue, while feasible in software switches is too complex to be implemented on the programmable hardware switch. Alternatively, the data plane can relay all flowkeys to the switch control processor or record all flowkeys in the data plane. However, these are not feasible; e.g., bandwidth between the data plane and the control plane is limited, and data plane memory is also limited. Some sketch constructions store heavy flowkeys together with the counters [11, 32, 46]. However, these are infeasible at line-rate on today's RMT switches.[3]

To reduce the memory use, prior work proposed an optimized baseline—when a packet arrives, it checks whether the frequency of a flowkey has exceeded the threshold by querying the sketch counter, and if so, it reports the key to the control plane [33, 39, 41]. Unfortunately, this still has a problem as "heavy" flowkeys may be reported redundantly every time a packet arrives and needs more control plane bandwidth. To avoid duplicate reporting to the control plane, we could use a Bloom filter to check if a heavy key has already been reported [33]. However, we need to configure the Bloom filter (i.e., bitmap size and number of hash functions) to have really low false positives since a false positive in the Bloom filter for the duplicate check is a potential miss of a heavy flowkey. Fig. 6c confirms this trade-off; we can configure the Bloom filter depending on the target miss rate and we find the memory footprint is correspondingly higher (We use 3 hash functions for Fig. 6c). Using a Bloom filter might be a valid approach if we allow some missing heavy flowkeys, we argue a design that targets a zero miss rate is more desirable.

We implement four possible strawman solutions to report heavy flowkeys and run microbenchmarks on a Tofino hardware switch to understand a trade-off between the accuracy and the resource consumption. Table 1 summarizes our analysis and shows that we have an undesirable trade-off between the miss rate of heavy flowkeys, data plane resources (mem-

[3] Specifically, HashPipe [46] cannot be directly implemented on RMT architecture due to complex memory access patterns (see [11] for more details). Precision [11] requires recirculation, which means some packets must go through entire pipeline again.

	Miss rate	CP bandwidth	DP resource
Recored every key in the DP	Zero	Low	Infeasible
Report every key to the CP	Zero	Infeasible	Low
Report heavy keys to the CP	Zero	Infeasible	Low
Report non-duplicate heavy keys	Low	Low	High

Table 1: Strawman solutions for tracking heavy flowkeys (CP: control plane, DP: data plane).

ory for keys and hash calls for Bloom filters), and the control plane bandwidth (for reporting keys).

B4. Pipeline stages: So far we have implicitly assumed that the switch has a single pool of resources on the switch (i.e., SRAM/TCAM, SALUs, and hash calls) that can be allocated to the sketch operations. In reality, the resources are partitioned across the pipeline stages. This impacts resource use in two ways. First, before an operation can be assigned to a stage, all required resources need to be available on that stage. If that is not the case, it needs to be moved to the next stage. Second, if there is a dependency between two operations, e.g., $O_1 \rightarrow O_2$ in the code, then O_2 must be placed on a later stage than O_1, even if there are unused resources available on stages earlier in the pipeline. For example, the sequential `if` clauses used by UnivMon (Fig. 4) create sequential dependencies between the *if* clauses.

This means that, depending on resources required by operations and dependencies between them, the compiler will only be able to use a subset of the resources on the switch. To account for this, we consider pipeline stages as a separate resource. Fig. 6d shows the number of pipeline stages needed as a function of level L if we respect this architectural constraint. We see that UnivMon requires similar or more pipeline stages than R-HHH with same configuration parameters and the gap is increasing as the number of levels increases. This is a direct result of the sequential dependencies in UnivMon. The number of pipeline stages used is measured by running the RMT resource mapper.

4 Optimizations

Next, we present a series of optimizations to address the resource bottlenecks we identified earlier. For each optimization, we discuss the key idea, before discussing the correctness and applicability constraints. Some of these optimizations (e.g., O1, O3, O4) have appeared in earlier theoretical efforts and demonstrated in other settings (e.g., FPGA, software switch). Our contribution here is translating these ideas to hardware switches. Others (O2, O5, O6) are novel to the best of our knowledge. As summarized in Table 2, our optimizations can be applied to a broad spectrum of published sketches for telemetry and benefit 15 out of the 19 sketches listed. Some sketches that are outside our scope cannot be supported as they either use: (1) processing logic that is infeasible in hardware (i.e., Hashpipe); (2) counter data structures different from sketches (i.e., BeauCoup); or (3) complex processing patterns such as packet recirculation (i.e., Precision).

Sketch Type	Sketch Name	Feasible on HW?	Applicability of SketchLib
Frequency Estimation / Heavy Hitters	Count-Min [21]	Yes	O6
	Count Sketch [17]	Yes	O1, O6
	MRAC [37]	Yes	O3, O5
	Hashpipe [46]	No	N/A, due to infeasible logic
	Precision [11]	Yes	No, uses packet recirculation
Hierarchical Heavy Hitters	RHHH [12]	Yes	O1, O2, O5, O6
	HHH [20]	Yes	O1, O6
Cardinality	PCSA [25]	Yes	O3, O5
	MRB [23]	Yes	O3, O5
	LogLog [22]	Yes	O3
	HyperLogLog [26]	Yes	O3
Entropy	EntropySketch [38]	Yes	O1
Change Detection	K-ary [36]	Yes	O1, O2, O6
Super Spreaders	SpreadSketch	Yes	O3, O5
	BeauCoup [18]	Yes	No, non-counter based sketch
General	UnivMon [41]	Yes	O1, O2, O3, O4, O5, O6
	FCM [47]	Yes	O6
	SketchLearn [32]	Yes	O2
	ElasticSketch [53]	Yes	Not applicable

Table 2: Applicability of SketchLib on existing sketches.

4.1 Optimizing Hash Calls

Both single- and multi-level sketches need to compute multiple hash functions, resulting in high hash call usage in the hardware pipeline. We describe two optimizations: consolidating short hash calls and reusing hash calls.

Optimization 1. Consolidate many short hash calls. We observe that many hash calls only need short-length (e.g., 1-bit) hash results. For instance, count sketch (Fig. 1) computes a series of 1 bit hash calls, s_1 to s_R. Similarly, UnivMon (Fig. 2 (b)) computes h_1 to h_L. We can reduce the number of hash calls by consolidating many short hash calls, as long as the inputs to the hash calls are the same.

Consider a count sketch with $R \times W = 3 \times 512$ counters. Per row, we need two hash results: a 9-bit (i.e., $\log_2 512 = 9$) hash to index into the counter array and a 1-bit hash for the "sign" of the counter. Instead of using $3 \cdot 2 = 6$ hash calls, we can instead use one hash call that returns a 30-bit result to provide the 6 hash calls as in Fig. 7. Note that splitting a long hash result only needs simple hardware shift and bit mask operations. R-HHH and UnivMon are also benefited as they use multiple count sketches. Further, UnivMon uses many 1-bit hash calls in its sampling stage.

Correctness and applicability: For this optimization to be valid, the split short-bit hash results from the longer hash result must use the same flowkey as the input and, if required by the sketching algorithm, be pairwise independent [17]. Independence is achieved by randomly picking (different) seeds for hash calls in practice [12, 41, 53]. Theoretical anal-

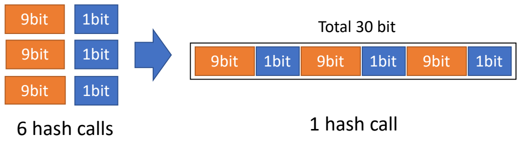


Figure 7: Optimization 1 reduces hash calls for count sketch.

Flowkey	Seed	Additional Condition	Opt
same	diff.	Sum of hash bit length is less than max capacity	O1
same	same	-	O2
diff.	same	One level of hash calls is executed	
diff.	diff.	-	-

Table 3: Conditions for optimization 1 and optimization 2.

ysis in other contexts [24, 35] shows that using different bits from the same hash call can also provide independence. Empirically, recent work [51] shows no accuracy loss for Univmon and our results (§6) confirm this with other sketches listed in Table 5. In addition, hash calls need to be short so that the sum of hash bit length is less than the length of one call (e.g., 32 bits). Fortunately, many single- and multi-level sketches [12, 17, 23, 25, 26, 37, 41] satisfy this condition.

Optimization 2: Reuse the hash calls across levels for multi-level sketches. Our second insight is that we can reuse the hash calls if there are no independence requirements across them ; i.e., they can use the same seed. Although hash independence is usually required across different counter arrays within a single level sketch, it is not required *across* levels [16]. Thus, we can use the *same* hash seed cross different levels for multi-level sketches.

Specifically, the original implementations of R-HHH and UnivMon (see Fig. 2) use a *different* hash seed in each of the *CS_level_i* count sketch executions. We can modify the code to reuse the same hash seed and reuse hash results when independence is not needed. This optimization reduces the number of hash calls significantly. For example in Fig. 2, R-HHH and UnivMon each have a set of hash calls F_i as $\{f_{i1}, f_{i2}, ... f_{i(2R)}\}$ at each level i of count sketch, resulting in $L \cdot 2R$ hash calls. By simply changing all of F_i to F_1, we reduce hash call usage from $L \cdot 2R$ to $2R$. For R-HHH, the result of F_1 is used to update one selected level of count sketch, and for UnivMon, result of F_1 can be used to update potentially multiple levels per packet.

Correctness and applicability: Reusing seed values across levels does not affect the theoretical independence requirements [16]. We empirically confirm in the evaluation that this optimization achieves similar accuracy (§6.1).

Table 3 summarizes the conditions under which the two hash optimizations are used. Note that for O2, if different levels' hashes have diverse output bit-length requirements, the hash call with the longest output bit-length will be used to supply hash results with various bit lengths. Also we need to make sure that the hash seeds are either the same in the first place or can be set to the same for O2 to apply.

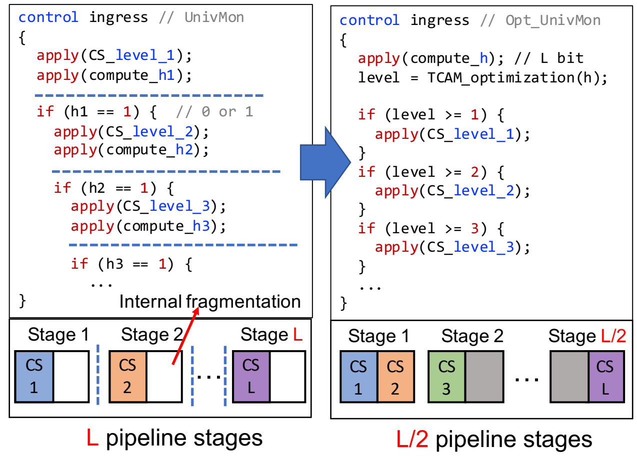


Figure 8: Optimization 3 removes the sequential computation dependency and reduces the usage of pipeline stages.

4.2 Optimizing Pipeline Stages

The sequential `if` clauses are observed in both single and multi-level sketches. This creates sequential compute dependencies and entails high usage of pipeline stages.

Optimization 3: Avoiding the sequential `if` clauses using a longest prefix match. To explain this optimization, we use UnivMon (Fig. 8) as an example. Deciding which levels to be updated for each flowkey creates a logical *dependency* between levels. Specifically, level i+1 needs to be updated only if the value of h_i returns 1 for hash functions $h_i : [n] \rightarrow \{0/1\}$. These L-level dependencies lead to an implementation as Fig. 8-left using sequential `if` clauses with hash values $(h_1, h_2, \ldots, h_L)$.

To address this bottleneck, our insight is that the number of leading 1-bits in $(h_1, h_2, \ldots, h_L)$ represents the sequence of "true" conditions in the `if` clauses. We observe that this is equivalent to the *longest prefix match* (LPM), which can be computed efficiently in hardware. That is, we can compute L hash bits together using a single L bit hash and use LPM to identify which layers need to be updated. This LPM operation is realized via TCAM as shown in Fig. 9. We insert rules with 1- and wildcard bits corresponding to each level and perform LPM to obtain the last level of UnivMon for each flowkey. LPM is relatively cheap—can be done in one pipeline stage using a small amount of TCAM. With this optimization, we can reduce the usage of pipeline stages by half if one count-sketch consumes half of the resources in one pipeline stage (Fig. 8-right).

Correctness and applicability: Our refactored implementation has the same functionality, resulting in the same updates to the sketch arrays. This optimization applies to many single and multi-level sketches that build on the power-of-two choices observation [23, 25, 26, 37, 55].

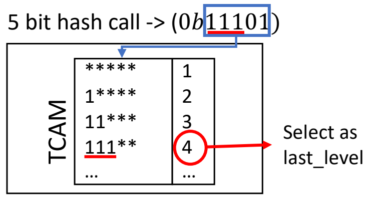


Figure 9: Replacing the sequential `if` clauses via TCAM.

4.3 Optimizing Memory Accesses

Sketches require memory accesses for their counter updates, leading to high SALU usage. This becomes especially significant for multi-level sketches.

Optimization 4: Refactor multi-level sketches to update one level per packet. We refactor multi-level sketching algorithms and their code to guarantee only one level is updated per flowkey. Recall that UnivMon needs to update one or more levels of count sketch (CS) for each packet with its flowkey. In Fig. 10 (top), a flowkey of packet K_{green} updates three levels, K_{gray} updates two levels, and K_{red} updates all levels of count sketch. Instead, our modified algorithm is guaranteed to update only the "last" level for each packet, as shown in Fig. 10 (bottom). The modified algorithm becomes structurally similar to other multi-level sketches that natively update only one level [12, 23, 25, 37]. As a result, the processing overhead is significantly reduced.

This "update-last-level" idea was also proposed to optimize UnivMon for embedded platforms [52] and software switches [40, 51]. Our contribution here is: (1) to extend this to programmable switches and (2) to generalize the idea to support updating arbitrary levels. Based on the algorithmic design, different multi-level sketches may require different optimization strategies to update a level (e.g., RHHH [12] modifies HHH [20] by randomly selecting a level to update). To implement this optimization, we can insert user-defined ternary rules in TCAM (as O3) to classify packets into different levels in a multi-level sketch.

Correctness and applicability: By construction, our modified algorithm provides equivalent functionality as the original version. As shown on the right side of Fig. 10 with K_{green} flowkey as an example, Levels 1 and 2 do not need to be updated anymore. Level 3 has the estimated flow count for this particular flow with the same or better accuracy since Level 3 only processes a smaller amount of traffic than Levels 1 and 2. Thus, the estimated count of K_{green} from Level 3 can be reused for Levels 1 and 2. This applies to all other flowkeys during the offline estimation in the network control plane.

To apply this optimization, a multi-level sketch should meet two conditions: (1) the original algorithm has multiple sketch updates per packet, and (2) it is algorithmically correct to reduce the multi-level updates to one per packet. That said, we acknowledge that there are scenarios where this optimization is not directly applicable. For instance, it is not obvious if/how we can refactor some multi-level sketches such as SketchLearn [32] to update only one level per flowkey (if possible). This requires future research.

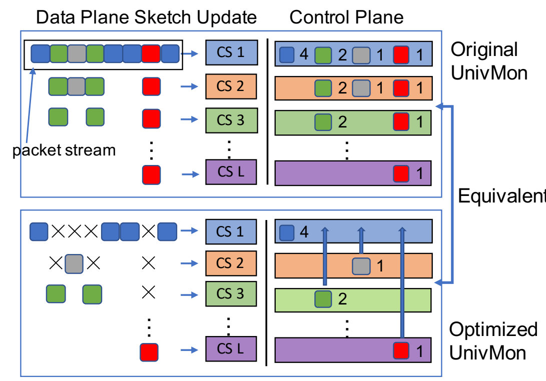


Figure 10: UnivMon updates only the last level per packet. CS stands for Count-Sketch.

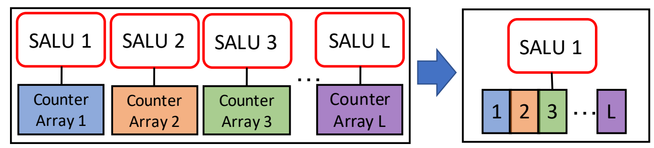


Figure 11: Optimization 5 removes unnecessary allocated SALUs by rewriting P4 code.

Optimization 5: Remove unnecessary SALU operations. A multi-level sketch maintains multiple independent levels of sketches. For each counter at each level, the compiler statically allocates an SALU for memory access. This results in high SALU usage, even if only one level needs to be updated per packet; i.e., usage is the same as updating all levels.

We can remove unnecessary SALUs when only one update is needed per packet. The reason why the compiler inefficiently preallocates SALUs for all possible memory accesses is that it is difficult to automatically figure out that only one update is needed at runtime. Our optimization restructures the P4 code to make this explicit for the compiler that only one count sketch update is needed per level. Instead of using separate counter arrays located in different switch memory regions, we consolidate the counter arrays of all levels in a single array located in one region of memory. This is possible because SALU can support random access, thus based on the selected level, we can compute the corresponding index value to access the consolidated register. Fig. 11 illustrates this SALU optimization. This optimization reduces the SALU requirements for multi-level sketches by a factor of L (the number of levels, e.g., 25 for R-HHH [12]).

Correctness and applicability: This technique does not affect accuracy because the modified code has the same functionality as the original version. We can apply the optimization to multi-level sketches that have the property of updating only one level per flowkey. There are many multi-level sketches satisfying this property [12, 23, 25, 37, 41].

Resource Bottlenecks	Optimizations	API
Hash Calls	O1. Consolidate short-bit hash calls	`hash_consolidate_and_split()`
	O2. Reuse hash calls across levels	`select_key_and_hash()`
Pipeline Stages	O3. Remove sequential if clauses using TCAM	`tcam_optimization()`
SALUs	O4. Update only one level per flowkey	-
	O5. Rewrite P4 code to reduce memory accesses	`consolidate_update()`
Resources for tracking heavy flowkeys	O6. Use a hash table to remove duplicate flowkeys	`heavy_flowkey_storage()`

Table 4: The relationships among the bottlenecks, optimizations and API calls.

4.4 Optimizing Heavy Flowkey Reporting

Optimization 6: Use a hash table and an exact-match table for checking duplicate flowkeys. As discussed in §3.2 B3, prior efforts [33, 39] use Bloom filters as the duplicate checker but the false positives from the filters will cause misses of heavy flowkeys, unless a very large Bloom filter is used. To improve this tradeoff between miss rate and data plane resource, we use a hash table and an exact-match table to check duplicates. Specifically, the hash table stores heavy flowkeys and detects whether there is a collision. For each heavy flowkey, if it is already stored in the hash table or exact-match table, it will not be reported to the controller; otherwise, it will be inserted to the hash table. But if this flowkey collides with another key in the hash table, then it will be reported to the controller which then inserts this flowkey to the exact-match table to filter future duplicate keys. In this way, we can ensure a zero miss rate on reporting heavy flowkeys.

Correctness and applicability: This optimization ensures a zero miss rate of heavy flowkeys because when collisions happen in the hash table, the flowkeys are reported to the control plane and inserted to the exact-match table (as a secondary duplicate checker). No unique heavy flowkeys are dropped in this mechanism. Compared to Bloom filters, this approach adds some additional control plane bandwidth when collisions happen in the hash table. As we evaluate in §6.5, this added bandwidth is small (e.g., 2% increase). This optimization can be applied to both single- and multi-level sketches requiring heavy flowkey tracking [12, 17, 41].

5 SketchLib API

In this section, we present our P4 API for helping sketch developers to use our optimizations. For each API call, we show the implementation for the macro and how the macro is used. SketchLib API supports both P4-14 and P4-16 [6]. Table 4 maps the optimizations to the API calls.

`hash_consolidate_and_split(Key,Seed,List(BitLen),BL_sum,List(Mask))`[4] reduces hash calls by consolidating small bit hash calls (O1). Fig. 12 shows how a sequence of short hash calls is replaced by a macro that uses only a single hash call with length the sum of all `BitLen` of the shorter hashes. The resulting hash value is then partitioned in shorter hashes. For P4-14, we split the result using `modify_field_with_shift(dst, src, shift, mask)` primitive (i.e., `dst = (src » shift) & mask`) where `mask` is a series of 1's with `BitLen` as shown. For P4-16, the same principle is applied, but bit slice operation (e.g., `h[BL1:0]`) is used. Note that the macro specifies both the number of short hashes being merged (*List*) and the names of the short hashes, so multiple macros must be defined if O1 is applied multiple times.

```
h1 = hash(sIP, seed1, 5);
h2 = hash(sIP, seed2, 3);
h3 = hash(sIP, seed3, 4);
```

```
hash_consolidate_and_split_3
  (sIP, seed1, 5, 3, 4, 12,
  0b11111, 0b111, 0b1111)
```

```
#define
  hash_consolidate_and_split_3
  (Key, Seed, BL1, BL2, BL3,
   BL_sum, mask1, mask2, mask3)
h = hash(Key, Seed, BL_sum);
h1 = h & mask1;
h2 = (h >> BL1) & mask2;
h3 = (h >> (BL1+BL2)) & mask3;
```

Figure 12: `hash_consolidate_and_split()`

`select_key_and_hash(List(Key),Level,Seed,BitLen)` implements O2 for the case one of the several hash calls with different `Key` and same `Seed` is selected for execution. Here, we can select the key in advance and use only one hash call to get the result as in Fig. 13. For instance, R-HHH can be optimized by using this API call. The example shown is a single hash call, but if multiple are needed (e.g. sketch with R =5 needs 5 hash calls), the number of hash calls can be increased. For the sketches that share the same `Key` and `Seed` (e.g., UnivMon), no separate API call is necessary since the hash value can simply be reused.

```
if (V == 1)
   h = hash(key1, seed, 3);
if (V == 2)
   h = hash(key2, seed, 3);
if (V == 3)
   h = hash(key3, seed, 3);
```

```
select_key_and_hash_3
  (key1, key2, key3, V, seed, 3)
```

```
#define select_key_and_hash_3
  (key1, key2, key3, V, Seed, BL)
if (V == 1)
   k = key1;
if (V == 2)
   k = key2;
if (V == 3)
   k = key3;
h = hash(k, Seed, BL);
```

Figure 13: `select_key_and_hash()`

`tcam_optimization(Hash_Result)` implements O3 to remove sequential `if` clauses by applying an equivalent a LPM table which uses TCAM to which levels need to be updated. The macro implements the use of the TCAM to look up the level (see Fig. 8).

`consolidate_update(Level,Index)` implements O5 to reduce memory accesses, as illustrated in Fig. 14. `Level`

[4] While there is no concept of `List` in P4, we use it to describe the type of parameters conceptually throughout this section. In our API implementations, it is converted to multiple parameters; e.g., `List(BitLen)` → `(BL1, BL2, BL3)` as shown in Fig. 12.

indicates the selected counter array and `Index` references the location for the memory update within the counter array. The API call consolidates counter arrays and computes the new address for the consolidated array. `size` indicates the bit length (e.g., 10) of the width (e.g., 1024).

```
if (V == 1)
  update_array_1(V, index);
if (V == 2)
  update_array_2(V, index);
if (V == 3)
  update_array_3(V, index);
```

```
consolidate_update_3(V, index)
```

```
#define consolidate_update_3
                           (V, index)
n_index = ((V-1)<<(size))+index;
update_array_1_to_3(n_index);
```

Figure 14: `consolidate_memory_update()`

`heavy_flowkey_storage(Key,List(Estimate),Threshold)` reduces the memory space for heavy flowkeys (O6). The challenge is checking whether the estimated flow count is above a threshold entirely in the data plane. Specifically, this entails computing the median value based on an estimated flow count from each row and comparing it to the threshold value. However, computing the median is not supported in the data plane. Instead, we leverage the fact that we can check whether the median of a set of values exceeds a threshold without computing the median as follows. We compare all of estimated flow count for all rows, as shown in lines 3-9 in Fig. 15 which is for $R = 3$ case. Then, the condition (sum (s1, s2, s3) $\geq$ 2) at line 11 is equal to (median(est1, est2, est3) > T).[5] This can be generalized for different Rs. We implement the duplicate filter using a hash table and a exact-match table. If a flowkey collides with an entry in the hash table and the exact-match table does not have an entry for the flowkey, we report it to switch control plane via a PCIe channel. Upon receiving the reported key, the switch control plane CPU adds entries into the exact-match table.

6 Evaluation

In this section, we evaluate the benefits of SketchLib on seven sketches. Across a range of settings, we see that SketchLib can reduce the resource footprint of sketches on switch hardware (up to 96%) while achieving similar accuracy.

6.1 Experimental Setup

Sketches. We implement all 15 sketches in Table 2 using SketchLib and source codes for sketches are available at [6]. Among 15 sketches, we pick seven representative sketches for our evaluation as in Table 5.

Testbed. We evaluate SketchLib on a local testbed with an Edgecore Wedge 100BF Tofino-based programmable switch and a server equipped with dual Intel Xeon Silver 4110 CPUs, 128GB RAM, and a 100Gbps Mellanox CX-4 NIC connected to the switch. We use the P4-16 version of SketchLib with Tofino SDE version of 9.1.1 in our experiments.

[5]For Count-Min sketch [21], we can use (sum (s1, s2, s3) $\geq$ 1).

```
01:#define heavy_flowkey_storage_3
                         (Key, Est1, Est2, Est3, T)
02:
03: s1, s2, s3 = 0;
04: if (Est1 > T)
05:   s1 = 1;
06: if (Est2 > T)
07:   s2 = 1;
08: if (Est3 > T)
09:   s3 = 1;
10:
// above threshold test
11: if (s1 + s2 + s3 >= 2) {
12:   if (HT[h(Key)] == empty) { // HashTable
13:     HT[h(Key)] = Key;
14:     send_to_cpu(Key);
15:   } else if(HT[h(Key)] != Key) {
16:     if (!(flowkey in MT)) { // MatchTable
17:       send_to_cpu(Key);
18:     }
19:   }
20: }
```

Figure 15: `heavy_flowkey_storage()`

Traces. We use five CAIDA backbone traces capture at 3/20/14 to 6/19/14 Sanjose, 1/21/16 Chicago, 5/17/18 to 8/16/18 New York City [7]. We split one hour traces into 30 second epochs. Each epoch includes about 12M-23M packets, with 398K distinct source IPs, 280K distinct destination IPs, and 1.6M distinct 5 tuples.

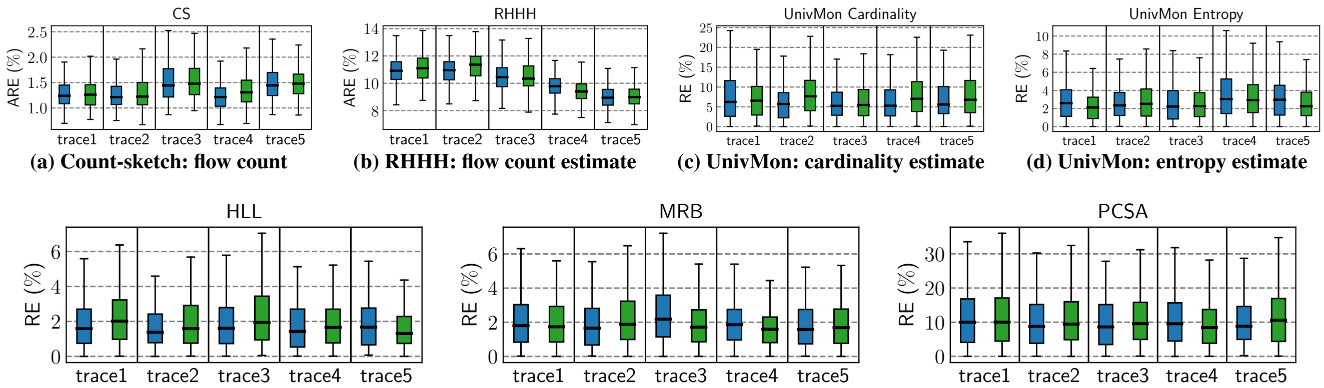
	Level (L)	Row (R)	Width (W)	Space
CS [17]	-	5	4096	80KB
HLL [26]	-	-	2048	8KB
UnivMon [41]	16	5	2048	640KB
R-HHH [12]	25	3	2048	600KB
MRAC [37]	12	-	2048	96KB
MRB [23]	16	-	4096	8KB
PCSA [25]	32	-	20	0.125KB

Table 5: Sketch parameters for evaluation.

Sketch parameters. Table 5 shows the configuration parameters for the sketches. Most sketches use 4 byte counters. The cardinality estimators (e.g., MRB and PCSA) use bitmap thus each counter is 1 bit.

Metrics. Depending on the sketch and the measurement task, we report two error metrics. For each metric, we run the experiment 5 times independently with different hash parameters and report the 25%, 50%, 75% percentiles of the errors. For brevity, we report results using source IP as the flowkey except for R-HHH, noting that the results are qualitatively similar for other types of flowkeys. R-HHH uses (source IP, destination IP) pair as flowkey as presented in the original paper [12].

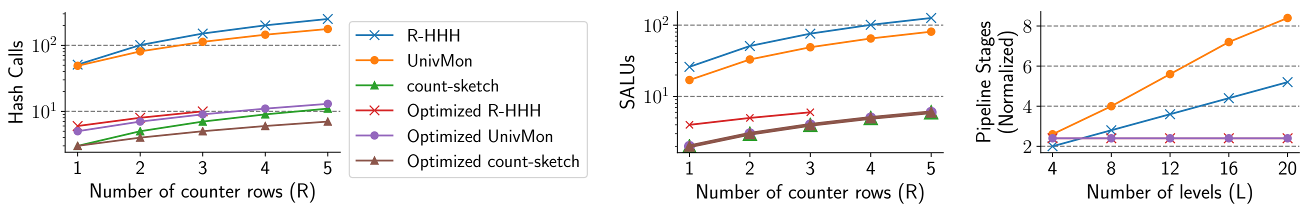
- Average Relative Error (ARE): $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i-\hat{f}_i|}{f_i}$, where k means the top k heavy flows. f_i is actual flow count for flow i and $\hat{f}_i$ is the estimated flow count from the sketch. $f_i \geq f_{i+1}$ for any i, thus it is sorted in descending order. We use k=50 for count sketch and R-HHH.
- Relative Error (RE): $\frac{|True-Estimate|}{True}$, where $True$ is ground truth value and $Estimate$ is estimated value. We use this metric for sketches that estimate cardinality and/or entropy.



(a) Count-sketch: flow count (b) RHHH: flow count estimate (c) UnivMon: cardinality estimate (d) UnivMon: entropy estimate

(e) HLL (f) MRB (g) PCSA

Figure 16: Accuracy comparison of sketches between original and optimized sketches across traces. Left: original, Right: optimized.



(a) Hash Calls (b) SALUs (c) Pipeline Stages

Figure 17: Resource consumption before/after optimizations.

6.2 Accuracy

We run the accuracy experiment of SketchLib in two ways. First, we show the accuracy is preserved between baseline software implementation and hardware implementation with SketchLib (§6.2.1). Second, we compare the accuracy of the hardware implementations with and without SketchLib (§6.2.2).

6.2.1 Comparison with the Software Baseline

Reporting methodology. We compare the accuracy of the sketch refactored with SketchLib (on hardware) against a baseline software implementation. The baseline software implementation runs sketches on the software. We run experiments over multiple traces with independent runs. After optimizing sketches with SketchLib, we run experiments on Tofino hardware with all five traces. For each one-hour trace, we randomly sample 40 30-second epochs and obtain 5 accuracy numbers per epoch with independent trials. The server replays traces to the switch using tcpreplay at a speed of 800K packets/second. Between epochs, we wait for switch control plane to pull counters and flowkeys from the data plane (see §7).

Result. Fig. 16 empirically validates that SketchLib optimizations achieve similar accuracy. For every trace, the left blue bar represents the software baseline and the right green bar is the hardware reported result with SketchLib applied.[6] Fig. 16a - Fig. 16d shows the accuracy of sketches that need to track heavy flowkeys and the rest show sketches that need to maintain only counter arrays. Fig. 16c and Fig. 16d show the errors of UnivMon for cardinality estimation and entropy estimation. We can visually confirm that the distributions of accuracy before and after optimizations are similar.

[6]We do not show MRAC as the estimation logic for MRAC is not public.

UnivMon	Without SketchLib			With SketchLib
Level (L)	8	6	5	16
Row (R)	4	5	6	5
Width (W)	32768	32768	32768	2048
RE (%)	95.4%	98.8%	99.4%	9.5%

Table 6: Relative error in cardinality estimation with and without SketchLib.

6.2.2 Accuracy Improvement with SketchLib

Reporting methodology. We want to compare the best accuracy between with and without SketchLib on the hardware. We use UnivMon for this experiment. To systematically sweep configuration parameters for the best accuracy without SketchLib, we exploit the property of UnivMon. Among three sketch parameters level (L), row (R), and width (W), L is the most critical parameter, thus we pick three highest feasible L. Then we find maximum R and lastly W. Given fixed L, we explored different parameter R other than maximum R but result was similar. We use the simulator with 40 samples of trace1. With SketchLib, we use the same configuration from the original UnivMon paper.

Result. Table 6 shows that all feasible configurations without SketchLib show high error rate more than 95%. On the other hand, UnivMon with SketchLib shows low error rate of 9.5%.

Sketches	Hash Calls (O1/O2)	SALUs	Pipeline Stages
CS	31%/0		9%
HLL	80%/0		86%
UnivMon	44%/47%	90%	65%
RHHH	32%/60%	92%	62%
MRAC	87%/0	91%	68%
MRB	90%/0	93%	76%
PCSA	92%/0	96%	86%

Table 7: Individual resource reductions by optimizations.

6.3 Switch Resource Consumption

Next, we report the resource usage improvements on the identified resource bottlenecks (Table 7). The sketch parameters used are reported in Table 5.

Reporting methodology. We measure resource usages from original implementation using RMT resource mapper and optimized implementation using Tofino compiler to measure resource reductions. To factor out resource reductions for different optimizations in Table 7, we wrote P4 code with individual optimizations applied using SketchLib APIs to measure the resource usages.

Hash calls. Table 7 shows that using O1 to consolidate the 1-bit hash calls is effective for both single and multi-level sketches. For example, the number of hash calls for count sketch is reduced by 31%. R-HHH and UnivMon benefit from O1 as they are composed of multiple count sketches. Further, PCSA, MRAC, MRB and HLL have a series of 1-bit hash calls which O1 improves. For UnivMon and R-HHH, we can apply both O1 and O2 by reusing hash calls across levels to further reduce hash calls by over 90%. We further investigate the sensitivity of the reduction of hash calls vs. sketch parameters in Fig. 17a.[7] Multi-level sketches UnivMon and R-HHH have significant reduction and the resource used is close to single-level count sketch.

Stateful ALUs. O5 applies only to multi-level sketches and reduces SALU usage significantly if there are many levels in the sketch. With 16-32 levels, O5 saves 92% to 96% of the SALUs. We can see in Fig. 17a that O5 reduces SALU of UnivMon and R-HHH significantly across rows.

Counter Memory Space. Interestingly, O5, which we designed to reduce SALU usage, additionally reduces memory space. Investigating this further, we find that original sketch implementations have a memory region fragmentation problem. One counter array is smaller than a block of SRAM, causing additional (unused) memory overhead per each counter array. O5 has the added benefit of consolidating counter arrays and achieve 54%–96% of resource reduction in memory space for multi-level sketches (not shown).

Pipeline stages. The reduction of pipeline stages depends on a combination of factors — hash calls, SALUs, and code dependencies. Table 7 shows reduced pipeline stages from 9% to 86% across sketches. Sketches where O5 applies (HLL, UnivMon, MRAC, MRB, PCSA) have a large reduction because it removes many sequential `if` clauses. Effectively, our optimization can make the footprint of multi-level sketches agnostic to number of levels (Fig. 17c).

[7]Missing point for R-HHH in Fig. 17 means it is infeasible.

	FCM native	SketchLib-optimized		
Resource	FCM+topK	FCM(+O6)	CM	UnivMon
Pipe. Stage	8	8	7	12
SRAM	9.5%	10.8%	8.0%	7.3%
TCAM	0%	0%	0%	0.3%
SALUs	20.8%	14.6%	14.6%	12.5%
Hash Calls	13.9%	9.7%	11.1%	18.1%
Hash Bits	5.6%	4.0%	4.0%	4.9%

Table 8: Comparison of hardware resource utilization.

		SketchLib-optimized		
	FCM+topK	FCM(+O6)	CM	UnivMon
HH (ARE)	1.41%	0.01%	0.13%	0.73%

Table 9: ARE of heavy hitter detection.

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.35%	0.84%	3.60%	6.15%	17.0%
SketchLib UnivMon	2.59%	2.08%	2.21%	2.36%	2.96%

Table 10: Entropy error (RE), FCM vs. SketchLib-optimized UnivMon.

6.4 Comparison with FCM

FCM [47] is a recently published sketch with general capability, and it is feasible on the programmable switch. Thus, we compare FCM against sketches optimized with SketchLib in terms of resource usages and accuracy. Table 8 shows resource utilization comparison between FCM and SketchLib optimized sketches. We use the same configuration from public FCM code [3], and make SketchLib-optimized sketch use similar resources to FCM.

Heavy hitter detection. Table 9 shows the accuracy result of heavy hitter detection. We can see that FCM+topK suffers from a high error rate because of an inefficient mechanism for tracking heavy flowkey (approximate topK implementation of ElasticSketch [53]). Note that if FCM deploys one of our optimizations for tracking heavy flowkeys, FCM+O6 reduces the error rate significantly from 1.41% to 0.01%. We use the simulator with 40 samples of trace1 and report median ARE.

Entropy and cardinality. Table 10 and Table 11 compare entropy and cardinality estimation accuracy between FCM+topK and SketchLib-optimized UnivMon. In the experiments, UnivMon reports top-200 heavy hitters per level. For entropy, UnivMon shows a relatively stable error rate (2∼3%) across workloads, whereas FCM is dependent on workloads and the error rate can go up to 17%. For cardinality, the error rate of UnivMon is moderately increasing [8], whereas FCM suddenly becomes unusable after 5M flows. This is because Linear Counting [50] is used to estimate cardinality in FCM.

6.5 Tracking Heavy Flowkeys

To evaluate the impact of O6, we consider three metrics: miss rate, control plane bandwidth, and data plane memory. We

[8]We observe that, when UnivMon reports more heavy hitters per level, the cardinality error rate decreases (e.g., 17.58% in 10M flows with top-1000).

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.004%	0.107%	0.519%	100%	100%
SketchLib UnivMon	21.9%	20.7%	31.7%	39.5%	73.8%

Table 11: Cardinality error (RE), FCM vs. SketchLib-optimized UnivMon.

Resource	UnivMon	With SketchLib UnivMon + NFs (L2, L3, LB, FW)
Pipe. Stage	12	12
SRAM	7.3%	38.6%
TCAM	0.3%	25.0%
SALUs	12.5%	12.5%
Hash Calls	18.1%	18.1%
Hash Bits	4.9%	11.2%

Table 12: Sketches are infeasible without SketchLib. With SketchLib, there are rooms for additional network functions (L2/L3 forwarding, L4 load balancer, and stateful firewall).

compare SketchLib-optimized approach vs. an "optimal" software solution. For this evaluation, we use two sketches (CS, UM) that track "heavy" flowkeys. For each 1-hr trace, we split it into epochs as before, and set a target threshold corresponding to the top 0.2 percentile of flow sizes (The results are independent of the threshold; this is to make the experiment concrete). Across different traces and sketches, SketchLib incurs zero miss rate, and at most 2% increase in control plane bandwidth (due to small number of duplicates), using less than 400KB of data plane memory overall (independent of the threshold, results not shown for brevity). To put this in context, a Bloom-filter based strawman for suppressing duplicates as discussed in §3 configured with the same memory use has a miss rate of 0.2%. Overall, this confirms that SketchLib offers a more practical alternative to the infeasible, inaccurate, and/or expensive strawman solutions from §3.

6.6 Other Benchmarks

Additional Network Functions. After optimized with SketchLib, sketches can even coexist with additional network functions such as L2/L3 forwarding, L4 load balancer, and stateful firewall. Table 12 shows resource utilization for additional network functions.

Code simplification. In addition to the resource efficiency benefits, our optimizations also simplify the sketch implementations by reducing the lines of code, as shown in Table 13.

Compilation time. We also measured compilation time to see whether our modified code will add significant overhead to the compiler. We measure compile time is measured on the server specified in (§6.1). For most cases, there was a negligible ($\leq$ 1 second) increase (not shown).

7 Related Work

Programmable switches. The programmable switch architecture was introduced by Bosshart et al [15]. Subsequent work proposed a programming framework [14], functional hardware [1], and also compilation workflows [34]. Other vendors have developed programmable pipelined architectures and compilation workflows from P4 or P4-like primitives [4, 5]. While our focus is on Tofino, our approach could be useful for other platforms as well.

Sketch	CS	HLL	UM	RHHH	MRAC	MRB	PCSA
Before	201	290	460	471	261	317	305
After	131	112	127	128	91	94	93

Table 13: Lines of code simplification (UM stands for UnivMon).

Optimizing sketches. HashPipe [46] focused on heavy hitter detection, but is not feasible in the current hardware. Other work has focused on the optimizing sketching algorithms in software switches (e.g., [31, 40, 51]). However, some of their ideas do not translate into a hardware context. For instance, NitroSketch increases the memory footprint to reduce CPU consumption, but the key bottleneck in hardware is different. Similarly, other approaches split a sketch into a fast and slow path on the software switch (e.g., [31]). Unfortunately, this is not relevant in hardware since we need all operations to be in the fast path. Some recent work [51, 52] specifically focus on optimizing UnivMon for embedded platforms and software switches. We translate these insights to a switch hardware realization, and generalize beyond UnivMon.

Control plane reporting. While this work focuses on optimizing data plane components of sketch-based monitoring, there are other challenges in accurately retrieving sketch counters in the control plane. Naïvely retrieving the counters using the existing control plane APIs can result in poor accuracy due to a nonnegligible amount of read and reset delays. We analyze this problem and suggest recommendations in parallel work [44].

Other work in network telemetry. Our focus in this paper is on sketch-based telemetry. There are other efforts for complementary monitoring capabilities (e.g., [29, 30, 48]) and performance-oriented objectives (e.g., [28, 45]).

8 Conclusions

Given increasing traffic rates and rich telemetry required, we see the confluence of two trends: the use of sketching algorithms and programmable switch hardware. Unfortunately, existing sketch implementations are not efficiently realizable, thereby limiting their effectiveness and coexistence with other switch functions. To this end, we systematically analyze the resource bottlenecks, suggest correct-by-construction optimizations, and design a practical library to help developers use these optimizations. Our evaluations show that the SketchLib library is broadly applicable to many sketches and reduces their resource footprint while achieving similar accuracy.

This work focuses on a single sketch-based monitoring task written using SketchLib APIs. We plan to support multiple tasks on a switch and automate the optimizations by integrating our techniques with a compiler as future work.

Acknowledgement

We would like to thank the anonymous NSDI reviewers and our shepherd, Brighten Godfrey for their helpful comments. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF awards 1565343, 1700521, 2106946, and 2107086.

References

[1] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.

[2] Broadcom Trident 3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[3] FCM-sketch source code. https://github.com/fcm-project/fcm_p4.

[4] Marvell LiquidIO SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers.html.

[5] Netronome Agilio SmartNICs. https://www.netronome.com/products/nfe/.

[6] Open Sourced SketchLib. https://github.com/SketchLib.

[7] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml.

[8] $P4_{14}$ Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf, 2018.

[9] NPL Specifications . https://nplang.org/npl/specifications/, 2020.

[10] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., ET AL. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 503–514.

[11] BEN-BASAT, R., CHEN, X., EINZIGER, G., AND ROTTENSTREICH, O. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), IEEE, pp. 313–323.

[12] BEN BASAT, R., EINZIGER, G., FRIEDMAN, R., LUIZELLI, M. C., AND WAISBARD, E. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 127–140.

[13] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), pp. 1–12.

[14] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[15] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[16] BRAVERMAN, V., AND OSTROVSKY, R. Zero-one frequency laws. In *Proc. of STOC* (2010).

[17] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.

[18] CHEN, X., LANDAU-FEIBISH, S., BRAVERMAN, M., AND REXFORD, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 226–239.

[19] CLAISE, B. Cisco systems NetFlow services export version 9. RFC 3954.

[20] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference* (2003), Elsevier, pp. 464–475.

[21] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[22] DURAND, M., AND FLAJOLET, P. Loglog counting of large cardinalities. In *European Symposium on Algorithms* (2003), Springer, pp. 605–617.

[23] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 153–166.

[24] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[25] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[26] FLAJOLET, P., RIC FUSY, GANDOUET, O., AND ET AL. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA* (2007).

[27] GARCIA-TEODORO, P., DIAZ-VERDEJO, J., MACIÁ-FERNÁNDEZ, G., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1-2 (2009), 18–28.

[28] GHASEMI, M., BENSON, T., AND REXFORD, J. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research* (2017), pp. 61–74.

[29] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 357–371.

[30] HARRISON, R., CAI, Q., GUPTA, A., AND REXFORD, J. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research* (2018), pp. 1–7.

[31] HUANG, Q., JIN, X., LEE, P. P., LI, R., TANG, L., CHEN, Y.-C., AND ZHANG, G. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 113–126.

[32] HUANG, Q., LEE, P. P., AND BAO, Y. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 576–590.

[33] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP* (2017).

[34] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 103–115.

[35] KIRSCH, A., AND MITZENMACHER, M. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[36] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 234–247.

[37] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (2004), 177–188.

[38] LALL, A., SEKAR, V., OGIHARA, M., XU, J., AND ZHANG, H. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review* (2006).

[39] LIU, Z., BAI, Z., LIU, Z., LI, X., KIM, C., BRAVERMAN, V., JIN, X., AND STOICA, I. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST* (2019).

[40] LIU, Z., BEN-BASAT, R., EINZIGER, G., KASSNER, Y., BRAVERMAN, V., FRIEDMAN, R., AND SEKAR, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 334–350.

[41] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 101–114.

[42] METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory* (2005), Springer, pp. 398–412.

[43] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 15–28.

[44] NAMKUNG, H., KIM, D., LIU, Z., SEKAR, V., AND STEENKISTE, P. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *Proceedings of the Symposium on SDN Research* (2021).

[45] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 85–98.

[46] SIVARAMAN, V., NARAYANA, S., ROTTENSTREICH, O., MUTHUKRISHNAN, S., AND REXFORD, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (2017), pp. 164–176.

[47] SONG, C. H., KANNAN, P. G., LOW, B. K. H., AND CHAN, M. C. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 78–92.

[48] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 453–456.

[49] WANG, M., LI, B., AND LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Proc. of IEEE ICDCS* (2004).

[50] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS) 15*, 2 (1990), 208–229.

[51] XIAO, Q., TANG, Z., AND CHEN, S. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM* (2020).

[52] YANG, M., ZHANG, J., GADRE, A., LIU, Z., KUMAR, S., AND SEKAR, V. Joltik: enabling energy-efficient" future-proof" analytics on low-power wide-area networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (2020), pp. 1–14.

[53] YANG, T., JIANG, J., LIU, P., HUANG, Q., GONG, J., ZHOU, Y., MIAO, R., LI, X., AND UHLIG, S. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 561–575.

[54] YU, D., ZHU, Y., ARZANI, B., FONSECA, R., ZHANG, T., DENG, K., AND YUAN, L. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.

[55] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI* (2013).

[56] ZHOU, Y., ZHANG, D., GAO, K., SUN, C., CAO, J., WANG, Y., XU, M., AND WU, J. Newton: intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 295–308.

A Comparison of RMT resource mapper and Tofino compiler

To validate RMT resource mapper as a proxy for Tofino compiler, we conduct experiments to compare resource allocation results of RMT resource mapper and the Tofino compiler. We pick five different sketches (UnivMon, R-HHH, PCSA, HLL, and MRB). We vary one parameter of sketches while fixing other parameters and analyze the resource allocation results. We focus on five different resource types; pipeline stages, hash calls, SALU, SRAM, and TCAM.

Fig. 18–Fig. 22 illustrate the results. Note that all of the resource usages are normalized. We can see that for hash calls, SALU, SRAM, and TCAM usages are identical between RMT resource mapper and the Tofino compiler. For pipeline stages, results are the same for PCSA, HLL, and MRB. However, RMT resource mapper finds mapping which uses fewer pipeline stages than the Tofino compiler for UnivMon and R-HHH. RMT resource mapper minimizes stages while the Tofino compiler finds more sparse mapping (e.g., mapping a small number of tables per stage). We validate both of the mappings from RMT resource mapper and Tofino compiler are valid. We confirm with the vendor that the Tofino compiler uses complex heuristics and the cost function of power budget and compilation time, which are different from that of RMT resource mapper and can introduce the gap. Our extensions to the RMT resource mapper is available at [6].

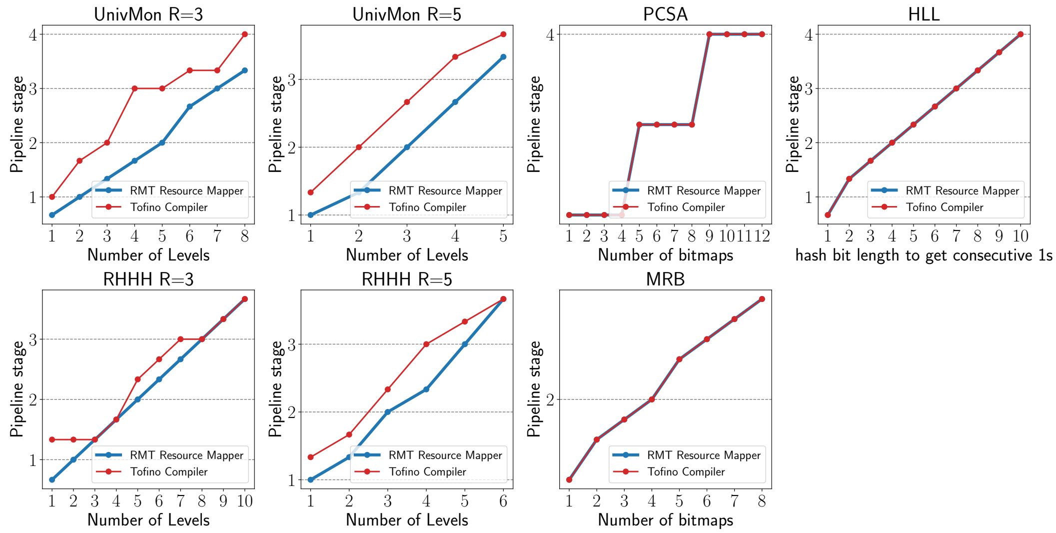


Figure 18: RMT resource mapper vs. Tofino compiler: pipeline stages

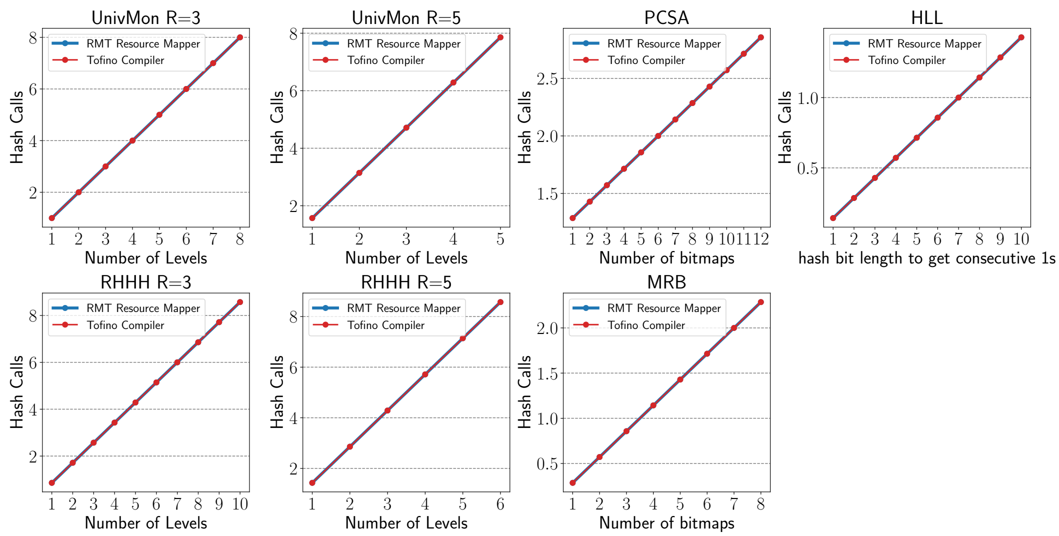


Figure 19: RMT resource mapper vs. Tofino compiler: Hash Call

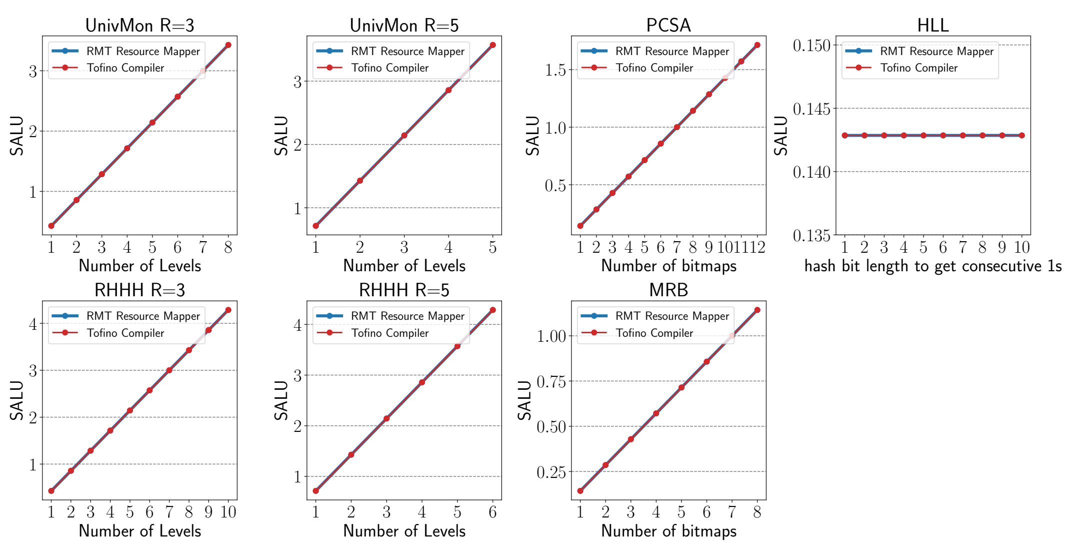


Figure 20: RMT resource mapper vs. Tofino compiler: SALU

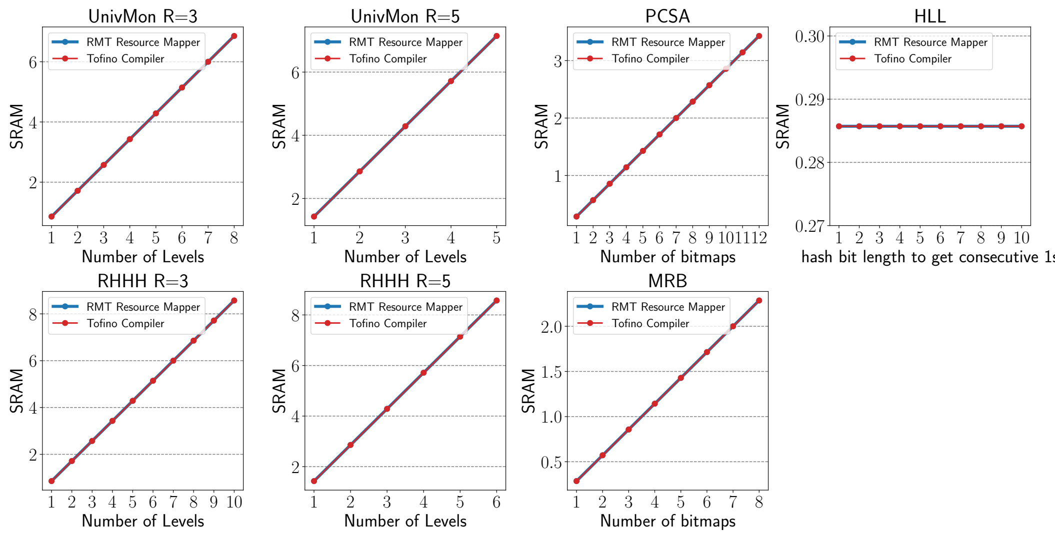


Figure 21: RMT resource mapper vs. Tofino compiler: SRAM

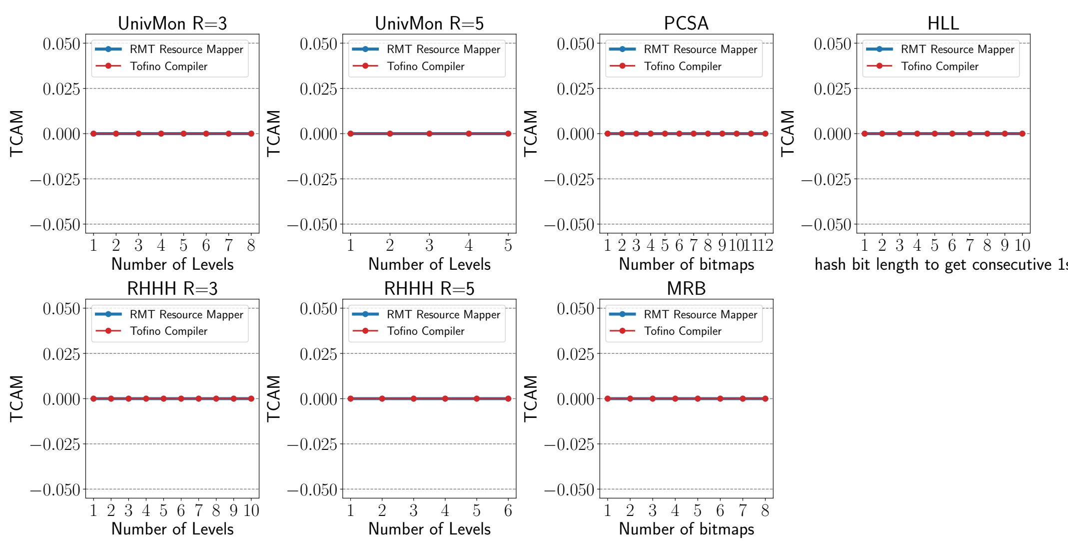


Figure 22: RMT resource mapper vs. Tofino compiler: TCAM