HeteroSketch: Coordinating Network-wide Monitoring in Heterogeneous and Dynamic Networks

Anup Agarwal
Carnegie Mellon University

Zaoxing Liu
Boston University

Srinivasan Seshan
Carnegie Mellon University

Abstract

Network monitoring and measurement have always been critical components of network management. Recent developments in sketch-based monitoring techniques and the deployment opportunities arising from the increasing programmability of network elements (e.g., programmable switches, SmartNICs, and software switches) have made the possibility of accurate, detailed, network-wide telemetry tantalizingly within reach. However, the wide heterogeneity of the programmable hardware and dynamic changes in both resources available and resources needed for monitoring over time make existing approaches to network-wide monitoring impractical.

We present HeteroSketch, a framework that consists of two main components: (1) a profiling tool that automatically quantifies the capabilities of arbitrary hardware by predicting their performance for sketching algorithms, and (2) an optimization framework that decides placement of measurement tasks and resource allocation for devices to meet monitoring goals while considering heterogeneous device capabilities. HeteroSketch enables optimized deployments for large networks (> 40,000 nodes) using a novel clustering approach and enables prompt responses to network topology, traffic, query, and resource dynamics. Our evaluation shows that HeteroSketch reduces resource overheads by $20-60\%$ compared to prior art, while maintaining monitoring performance, coverage, and accuracy.

1 Introduction

The ability to monitor network traffic in-situ and at-large-scale is a critical enabler for many networked management applications, including traffic engineering, load balancing, attack and anomaly detection, provisioning, and congestion control/fairness [1–7]. However, network-wide monitoring has proven to be challenging due to limitations on what measurements can be made and where these measurements can be taken. Recent developments in sketch-based monitoring and network programmability have made significant progress in eliminating these limitations and have made it possible to consider practical network-wide monitoring designs.

Sketch-based monitoring designs [8–13] demonstrate that sketches offer provable accuracy guarantees on a wide spectrum of metrics of interest using a small amount of memory and that independent sketch instances monitoring different parts of the network can be merged to obtain network-wide aggregated results. As a result, sketch-based monitoring has emerged as a promising alternative to traditional sampling-based monitoring tools (e.g., NetFlow [14] and sFlow [15]). The growing popularity of programmable network elements, such as programmable switches [16, 17], SmartNICs [18, 19], and software-switches [20, 21], have made it possible to deploy these sketch-based designs throughout a network – enabling highly-effective network-wide monitoring capabilities.

Despite significant recent progress [10–13, 22], we argue that deploying sketch-based monitoring in a network-wide setting remains impractical. The reason behind this is that existing network-wide solutions [11, 22, 23] assume an abstract network model without properly considering the *heterogeneity* and *dynamics* in the network. First, with growing types of programmable devices whose hardware architectures are dramatically different (e.g., ASIC, CPU, FPGA), it remains unclear how to deploy sketches among heterogeneous computation and memory hierarchies for optimized resource efficiency. Second, since monitoring capabilities share the same infrastructure with other network services [24–26] and monitoring requirements vary over time, the available and required resources for monitoring can change dynamically. We require an agile solution that can incorporate device heterogeneity and quickly adjust to network dynamics for best possible monitoring performance.

In this paper, we present **HeteroSketch**, a network-wide flow monitoring framework that coordinates sketch-based measurement to determine task placement and resource allocation for a network of heterogeneous devices. HeteroSketch has two main components: (1) a device characterization tool that automates quantified reasoning about the performance and resource usage of sketches on arbitrary de-

vice architectures and (2) an optimization framework that computes the placement of measurement tasks while considering available heterogeneous device resources and monitoring goals including forwarding performance, resource efficiency, monitoring accuracy, and flow coverage.

When designing our device characterization tool, we need to deal with a broad spectrum of programmable architectures such as CPU [20, 21], FPGA [19, 27], ASIC [16], and multi-core system-on-chip (SoC) [18]. Given the difficulty in accurately predicting the performance of arbitrary code under diverse workloads and hardware architectures [28, 29], we scope our efforts to sketches to create a practical solution. Our design is inspired by the observation that many sketches perform similar computations. We analyze the key operations of sketches and find that the performance of a sketch depends heavily on primitive operations – hash computations, counter updates, and random memory lookups. With that in mind, we then characterize these operations on current (and possibly new) hardware using automated micro-benchmarks, and leverage these measurements to express the performance and resource usage of a sketch. As evaluated in §7.1, our profiler accurately predicts sketch performance on programmable hardware with less than 6% mean relative error.

With precise performance profiles as input, HeteroSketch must address the tightly coupled monitoring goals, traffic demands, forwarding performance, sketch configurations, and resource usage, and the tradeoffs between them. Task placement and resource allocation requires a carefully crafted optimizer to incorporate the cost/benefit of different deployment options. We formulate a Mixed-Integer Program (MIP) to optimize resource efficiency while preserving forwarding performance, monitoring accuracy, and flow coverage.

Given the complexity introduced due to heterogeneous devices (non-convexity) and network scale, even a state-of-the-art solver [30] takes hours to solve the MIP, leading to stale solutions in face of network dynamics. We develop a clustering technique based on observed structures in network topologies and traffic patterns to partition the optimization into independent sub-problems. This allows HeteroSketch to scale to today's data center networks having tens of thousands of devices and respond to dynamics within a few seconds to a few minutes while maintaining near optimal allocations.

We implement HeteroSketch by porting state-of-the-art sketch implementations [8, 9, 11, 13] into representative programmable devices (Barefoot Tofino [16], Netronome Agilio SmartNIC [18], Xilinx FPGA NIC [27], and Open vSwitch (OVS) [20]) and encode the optimization in the Gurobi [30] solver. Our evaluation with more than 40,000 nodes demonstrates that our heterogeneity-aware optimization can achieve $20-60\%$ better resource efficiency (e.g., 50k CPU cores instead of 70k CPU cores) compared to prior solutions with the same performance, accuracy, and coverage while responding to network dynamics in a few seconds to a few minutes.

Contributions. We make the following contributions:

- We present HeteroSketch, the first system to our knowledge that performs coordinated sketch-based network-wide monitoring over a network of heterogeneous devices and caters to network dynamics. (§3)
- We develop a profiler that allows users to predict the performance of sketches on heterogeneous devices. (§4)
- We formulate a mixed-integer program to optimize sketch placement and resource allocation over heterogeneous devices and propose techniques to quickly optimize when topology, queries, traffic, or resources change. (§5, §6)
- We show that HeteroSketch is able to place tasks and allocate resources over network topologies, achieving greater scale and optimality than existing systems. (§7)

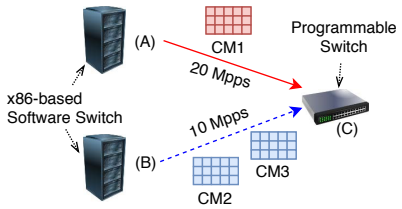
2 Background and Motivation

In this section, we describe how heterogeneous programmable data planes bring new opportunities and challenges to deploy network-wide monitoring under varying demands. We then discuss existing network-wide monitoring efforts.

Programmable Data Plane. Progress in programmable network devices is moving the network data plane towards a highly programmable infrastructure. This programmable infrastructure opens up the opportunity to develop measurement algorithms for a variety of fine-grained, flexible measurement tasks. For example, significant progress has been made in developing sketching algorithms [11, 12, 31] on the RMT architecture [32], where packets are processed over a series of reconfigurable match-action tables with user-defined actions in a pipeline. Similarly, multi-engine SmartNICs [18] consisting of a pool of general purpose processing elements (i.e., micro-engines) are a cost-effective option to allow hosts to offload monitoring capabilities or other parallel computation from CPU. These programmable devices enable the development of highly flexible and performant future-proof network-wide monitoring for various network demands.

New Requirements in Network-wide Monitoring. For a long time, network monitoring research has been focused on pursuing *accuracy* over other goals, such as forwarding *performance* with less computing and memory resources, and *scalability* by supporting larger-scale networks. While these requirements continue to be important, we believe that there are two significant roadblocks that make it difficult or impractical to use existing sketch-based designs at scale in future programmable data planes:

- *Heterogeneity in the network:* Network data planes are becoming increasingly heterogeneous with devices such as x86-based software switches [20], ASIC-based programmable switches [16], multicore system-on-chip (SoC) SmartNICs [18], and FPGA NICs [19, 27]. These devices are designed with diverse architectures and present very different resource bottlenecks to the programs that execute on them. The challenge lies in how to precisely character-



Strategy		Device			Total Resources
		A	B	C	
Network-wide UnivMon	Placement	CM1	CM2	CM3	
	Resources required	3 cores, 100 KB	2 cores, 100 KB	100 KB	5 cores, 300 KB
Better Strategy	Placement	-	CM2, CM3	CM1	
	Resources required	2 cores, 0 KB	2 cores, 200 KB	100 KB	4 cores, 300 KB

Figure 1: Example of network-wide UnivMon not optimally placing the sketches.

ize the performance of monitoring programs on current and possibly new devices and use these insights to optimize resource usage.

- *Network dynamics:* Network monitoring serves as a data collector and analyzer for other co-located network services (e.g., traffic engineering, load balancing, and anomaly detection). As all these services share the same infrastructure and their monitoring needs change, any network-wide monitoring system should quickly adjust to the following "network dynamics": (1) topology change, (2) monitoring query change, (3) traffic demand change, and (4) available resource change, in order to provide best monitoring performance and not interrupt other concurrent services.

Current Network-wide Monitoring and Limitations. In small networks, we can consider using techniques that record all packets or flows passing through the network for full accuracy (e.g., T-RAT [33], vCRIB [34], and OmniMon [35]). However, in practice, with the desire for *real-time* and *accurate* monitoring over large traffic volumes and dynamics, operators usually cannot afford to record all packets or flows on their network devices due to high resource usage and processing latency. Recently developed zoom-in techniques (e.g., Sonata [36] and ProgME [37], and others [38, 39]) provide no theoretical accuracy guarantee for dynamic traffic workloads. Systems such as NetFlow/sFlow [14, 15] and cSAMP [23] reduce overhead by recording only a fraction of packets via packet or flow sampling to compute measurement results. As shown in prior efforts [40, 41], these sampling approaches have low measurement accuracy in various tasks and workloads.

Sketching algorithms (sketches) address the drawbacks of sampling. At a high level, sketches [8, 9, 42] are probabilistic data structures that store a small summary of the input traffic. They allow a proven trade-off between the *accuracy* of supported queries and the *space* of the summary. Sketching techniques have efficiently supported various monitoring tasks including: heavy hitter detection [8, 9, 11, 12, 31, 43], traffic change detection [11, 44], anomaly detection [11, 22, 45], entropy estimation [11, 46, 47], counting distinct flows [11, 12]. Most sketches mentioned above can be *linearly merged* to obtain aggregated results with the same additive error guarantees [48]. For example, Sketch 1 measuring flow set A can be merged with Sketch 2 measuring flow set B (e.g., by addition of the two counter tables) to obtain statistics about a combined flow set $A \cup B$, as long as Sketches 1 and 2 share the same hash and memory configurations.

Scheme	Resource Overhead	Heterogeneity & Dynamics	Memory-Accuracy Tradeoff
cSAMP [23]	High	✗	Poor
vCRIB [34]	High	Limited	Poor
OmniMon [35]	High	Limited	Poor
UnivMon [11]	Medium	✗	Good
HeteroSketch	Low	✓	Good

Table 1: Summary of qualitative comparisons of existing schemes and our approach (HeteroSketch).

Unfortunately, existing sketch-based monitoring solutions don't consider heterogeneity and dynamics, which affects their resource efficiency and/or accuracy (Table 1). For instance, Figure 1 shows a simple scenario where network-wide UnivMon [11] does no optimally place Count-Min Sketches [8], resulting in using 5 cores instead of 4. In this setting, we have three devices (CPU A, CPU B, and programmable switch C). We want 1 Count-Min sketch (CM1) to monitor traffic between devices A and C, and we want 2 Count-Min sketches (CM2 and CM3) to monitor traffic between devices B and C. We also assume that the programmable switch can only fit one sketch in its share of switch resources for monitoring tasks. CPU A requires 2 cores for forwarding 20Mpps while CPU B requires 1 core for 10Mpps. The key decision is which sketch should go on the programmable switch. *Better Strategy:* If we place CM1 on the switch, both CM2, CM3 run on CPU B consuming 1 core for sketching (combined 20M sketch operations per second). *Network-wide UnivMon:* If we place CM2 (or CM3) on the switch, then CM1 must be placed on CPU A consuming 1 core for sketching and CM3 (or CM2) must be placed on CPU B again consuming 1 core for sketching, for a total of 2 sketching cores. Note, placing only one of CM2 or CM3 on CPU B consumes 1 sketching core as cycles are wasted busy-polling for packets for performance reasons. UnivMon produces this placement as it tries to balance memory load across devices.

3 System Overview

We describe the high-level design of HeteroSketch and highlight the key challenges that the design must address.

3.1 Problem Scope

HeteroSketch provides a "One Big Switch" abstraction to the user, wherein the user can specify monitoring require-

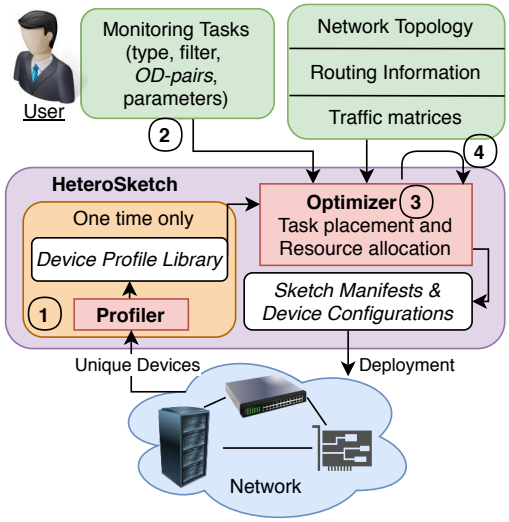


Figure 2: HeteroSketch Overview.

ments over all or a subset of traffic flowing between network endpoints or *origin-destination (OD)* pairs. Monitoring requirements include the type of the sketch, flow filter, and accuracy requirement (Figure 2). HeteroSketch takes these monitoring requirements and assigns particular sketch-based tasks among the components of a heterogeneous network of diverse devices using state-of-the-art sketching algorithms such as Count-Min [8], Count Sketch [9], and UnivMon [11].

This abstraction can be used to manage network monitoring in various settings. For example, HeteroSketch could be applied by Internet Service Providers (ISPs) to manage monitoring services internally or for their clients. In this paper, we envision cloud providers being early adopters of our network-wide monitoring system. In a multi-tenant cloud environment, a cloud provider would be able to offer monitoring-as-a-service to the tenants. Tasks corresponding to queries from different tenants would be placed within the network as opposed to just on the end-hosts that the tenant is using, i.e., NIC and switch resources can be indirectly accessed by tenants for monitoring purposes. The operator would make control decisions using a centralized view to address different measurement objectives. For instance, the operator could (1) manage monitoring requirements submitted by multiple independent tenants while incorporating any potential contention between monitoring tasks placed on the same device; and/or (2) deploy their own monitoring tasks (e.g., detecting compromised tenant VMs [49]). In any of the scenarios, our system makes it possible to load balance monitoring tasks between servers, NICs, and switches, and to prioritize resources such as CPU cores for other critical services and cloud applications.

3.2 HeteroSketch Workflow

As depicted in Figure 2, HeteroSketch has two main components: a performance profiler and an optimization framework.
Performance Profiler (① in Figure 2): For any new device, HeteroSketch needs to conduct offline performance characterization to add a new abstract profile into its device profile library. The device profile library allows HeteroSketch to predict the performance-resource trade-offs of different sketch configurations. We describe this Profiler in detail in Section 4.
Optimization Framework: Once HeteroSketch obtains user input as ② (monitoring requirements), the HeteroSketch Optimizer ③ outputs the configuration and mapping of sketches to devices (i.e., *sketch manifests* for each device) and the resources allocated to each device (i.e., *device configuration*). Based on the Optimizer output, HeteroSketch deploys sketches into the network and gathers network-wide statistics as other monitoring systems. If there are dynamic changes in user input, network topology, traffic demands, and available resources, HeteroSketch will perform a quick re-optimization ④. We describe this Optimizer in detail in Section 5.
Supported Queries: HeteroSketch currently supports sketch-based flow-level telemetry queries over flow sets defined over OD-pairs; e.g., heavy hitters, flow changes, entropy, distinct flows, among others, over flows across one or more OD pairs. For instance for network-wide heavy-hitters, a user can specify multiple OD-pairs to be monitored for the same query. HeteroSketch will then instantiate and collect (linearly merge) data from multiple instances of sketches (e.g., Count-Min and UnivMon) while ensuring that all OD-pairs are monitored and resource overhead is minimized. That said, telemetry queries that are not defined over flows, such as path-level queries in In-band Network Telemetry (INT) [50], or packet-level queries, which are not supported by sketches, are outside the scope of this paper.

3.3 Challenges and Key Insights

We describe the three main challenges that our design faces.

C1: [Heterogeneity] Predicting sketch performance for different resource allocations. Optimizing resource utilization requires characterization of exact costs and benefits of different deployment configurations. This is challenging because many characteristics of the program and the device architecture impact the processing time per packet.[1] For instance, devices may execute certain operations using ASICs and others using general purpose cores. The time for an instruction might depend on the allocated resources, e.g., memory access time depends on the working set. The program might have a complex, unpredictable control flow with many data dependencies. Past systems [28, 29] rely on low-level architecture specific counters and cache analysis to provide performance estimates. Such approaches do not provide the accuracy needed and would be difficult to generalize to other hardware.

Insight: We observe that (1) the primitive operations of sketches (e.g., hashing, memory updates) largely determine

[1] We represent performance in terms of time per packet or inverse throughput and use these terms interchangeably.

packet processing time, and (2) based on data flow analysis, most sketches have limited data dependencies and limited control flow. This means that there are enough independent operations to be performed (either for the same packet or across packets), that performance is broadly determined by the number of operations rather than their inter-dependencies. Therefore, we design a benchmark suite consisting micro-benchmarks of primitive sketch operations for a small set of sketch manifests. The Profiler composes these benchmarks to generate algebraic characterization of the resource-performance trade-off for arbitrary sketch manifests.

C2: [Formulation & Scalability] The optimization formulation of sketch placement over heterogeneous hardware results in a large and complex NP-Hard optimization problem. Despite using a state-of-the-art commercialized solver (e.g., Gurobi), it still needs order of hours to finish even for relatively small networks (≈1000 nodes). Specifically, the complex *non-convex* device profiles and scale of the network slow down the optimization.

Insight: We use the solver's advanced features (bi-linear constraints) to incorporate the non-convex device profiles. For scalability, we partition the optimization problem into disjoint sub-problems which are solved concurrently. We define these sub-problems by partitioning the network into clusters. We find that traditional clustering techniques such as spectral clustering [51] either result in infeasible sub-problems or solutions which are far from optimal (see §6). Our key insight is to define clustering affinity between nodes based on OD-pairs (traffic and monitoring requirements) rather than just network structure.

C3: [Dynamics] Sketch manifests and device configurations can become stale due to network dynamics including changes in monitoring requirements, available resources, logical topology, and traffic demands. A robust solution should adapt its deployment at the rate of these changes.

Insight: While the insights in C2 help scale the Optimizer to handle large topologies and bring down solving time from order of hours to a few minutes (§7.2), we supplement the clustering approach with a *Fast Path* that allows quicker responses (in a few seconds) to network dynamics. It leverages our observation that it is sufficient to recompute the placement/configuration for only a subset of devices which are *directly affected* by the network dynamics (§6.2).

4 Performance Profiler

We leverage the common structure of sketches to make the performance prediction problem tractable. As an example, we describe the structure of a canonical Count-Min sketch [8] that can be used for maintaining a summary of per-flow sizes.

The sketch maintains a counter table of rows and columns. On observing a <*key*, *value*> pair, a hash function is computed over the *key* for each row of the sketch. These hash values are used to index into the rows and the content of the corresponding cells is incremented by *value*. The updates to different rows are completely independent of each other, which is similar for a large set of sketches [8, 9, 11, 13, 42, 52]. With this structure, hash computation and memory update operations consume the majority of the time.

Our Approach. While the common structure of sketches allows us to manage the complexity introduced from the program's side, we still need to manage the complexity due to diverse devices. Specifically, for each device type, we have a three-phase approach to determine the sketch performance:

- **Phase 1:** Measure the time for primitive operations.
- **Phase 2:** Compose the time for different operations.
- **Phase 3:** Consider impact of device configurations.

Before diving into the details of the three phases, we provide a brief overview of the Profiler's operation and its setup.

Setup: The Profiler uses a three-device testbed consisting of the device being studied (or device under test, DUT), a sender, and a receiver. The three devices connected in a linear topology with the DUT configured to forward traffic from the sender to the receiver. Such a setup can be created without a lab environment or re-wiring, by changing forwarding configuration in a local or cloud deployment. A more detailed description of this testbed is provided in §7.1.

Overview: The Profiler treats the devices as "black-boxes" and makes few assumptions about the architecture. We assume that the DUT has a library to implement sketch manifests and that it exposes an API to allocate resources. The Profiler uses this API to study the DUT's forwarding rate for a limited set of sketch manifest and device configuration combinations. The Profiler does not need any code instrumentation, hardware counters, or precise time-stamping, it simply studies the end-to-end forwarding rate.

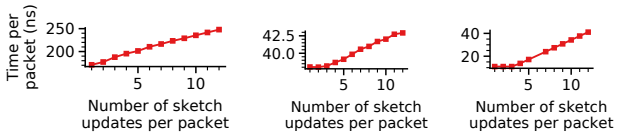
For each device, the Profiler models time per packet as an algebraic function of *sketch parameters*, *device parameters* and *device configuration*. The sketch parameters include counts for primitive sketch operations, which are obtained from the sketch manifest (e.g., sketch type, the numbers of rows and columns [8, 9], and the number of levels (sketch instances) [11]). We obtain device parameters from micro-benchmarks for the primitive operations. Device configuration specifies the resources, including memory, processor cores, micro-engines, switch stages/ALUs, lookup tables, flip-flops, and/or DSPs. We believe this approach generalizes to support architectures beyond the hardware at hand (Table 2).

4.1 Detailed Design of the Profiler

Phase 1: Primitive Operations. In this phase, we evaluate the time for the following primitive operations per sketch update: hash computations (compute capabilities), memory accesses (impact of memory hierarchy), coin tosses (random

Type	Hardware
CPU (Open vSwitch)	Intel® Xeon® Silver 4110 CPU @ 2.10GHz (32KB L1, 256KB L2, 8MB L3 cache) with Mellanox ConnectX 4 NIC [53][20]
SoC SmartNIC	Netronome Agilio® CX 1x40GbE
FPGA NIC	Xilinx® Alveo™ U280 Data Center accelerator card
Prog. Switch	Barefoot Tofino

Table 2: Devices tested with Profiler.



(a) CPU **(b) SoC NIC** **(c) FPGA NIC**

Figure 3: Phase 1 — Hashing micro-benchmark. (a) single core running OVS and sketching module. (b) 54 micro-engines. (c) single hash instance. Y axes show time per packet (ns).

number generation), and packet forwarding. For simplicity, we count the update to each row of a sketch as a separate *sketch update*, e.g., two Count-Min sketches with 4 rows each would make up 8 sketch updates per packet.

The Profiler studies the variation of time per packet as the configuration of a Count-Min sketch is varied. For hashing micro-benchmark (Figure 3), we vary the number of sketch updates for a small fixed amount of memory (to ensure memory does not become bottleneck). For the memory micro-benchmark (Figure 4a), we vary the sketch memory for a fixed number of sketch updates. We handle coin tosses similar to hashes but with sketches such as [13, 52] which rely on random number generation. Note, for all the micro-benchmarks, we generate flow keys that are uniformly distributed. This results in uniform memory access pattern for Count-Min sketch and allows us to estimate the worst case performance.

Phase 2: Composition. Given time for primitive operations from Phase 1, Phase 2 determines how different micro-benchmarks should be composed to obtain the total time per packet. The goal is to capture unique properties of how a device architecture combines the primitive operations by testing for three key properties: (1) Memory and compute concurrency, (2) Forwarding and sketching concurrency, and (3) Sketch access frequency. For each property, the Profiler defines a set of expected behaviors (composition functions). These correspond to different device architecture or sketch implementation choices. We currently rely on manually inspecting the micro-benchmarks to identify the device/implementation behavior. This process can be automated in a straightforward fashion. We detail the properties and behaviors below.

Memory and compute concurrency: Compute and memory operations in a system may be *coupled* or *decoupled* depending on the hardware: (1) in a *coupled* system, hashing and memory operations might contend for the same hardware units, (2) in the *decoupled* case, the memory and hash operations have zero contention, i.e.,

$$n_{sketch} = n_{compute} + n_{memory} \triangleright \text{coupled}$$
$$n_{sketch} = \max(n_{compute}, n_{memory}) \triangleright \text{decoupled}$$
$$n_{compute} = k_1 + u_h \cdot h$$
$$n_{memory} = k_2 + u_m \cdot T(m)$$

where n_{sketch} is the time for sketch updates, u_h is the number of hash computations per packet and u_m is the number of accesses to the sketch memory per packet. h is the time per hash computation, k_1 and k_2 are constants, and $T(m)$ is the time per memory access given m amount of total memory has been allocated for sketching. For the coupled case, the memory access benchmark would subsume the time for hashing and vice versa. In this case, $T(.)$ is used to represent *additional* time per memory access incurred due to potential cache misses. This is extracted by adjusting for the time per hash.

For CPU, we use the coupled model as hash computation has memory instructions which prohibit full overlap with sketch memory accesses. For the FPGA and SoC SmartNIC, we use the decoupled model, as their compute (hashing) units and memory units are completely disjoint.

For sketches which have multiple levels or control paths (e.g., UnivMon [11]), the number of primitive operations can be different for different packets based on their flow key as well as the sketch memory access pattern can be non-uniform even for uniformly distributed flow keys. In this case we interpret u_h, u_m as the expected number of operations per packet and use $T(\text{Effective uniformly accessed memory})$ instead of $T(\text{Total memory})$. For brevity we discuss the details of computing "effective uniformly accessed memory" in Appendix B. We find that UnivMon behaves as if at most 4 of its levels are accessed uniformly irrespective of the amount of sketch memory and across devices.

Forwarding and sketching concurrency: Packet forwarding and sketching can be done in parallel or in the same thread(s). If done in parallel, the time per packet would be the maximum of the inverse throughput of forwarding and sketching; otherwise, the sketching benchmarks would subsume the time for forwarding, i.e.,

$$N = \max(n_{fwd}, n_{sketch}) \triangleright \text{concurrent}$$
$$N = n_{sketch} \triangleright \text{sequential forwarding and sketching}$$

where N is the time per packet and n_{fwd} corresponds to the forwarding inverse throughput. For both SoC SmartNIC and CPU, sketching is done on the critical path (sequential). On the FPGA NIC sketching is off the critical path (concurrent).

Sketch access frequency: Not all sketches may be updated for every packet. For instance, the user may want certain sketches to only monitor a subset of packets forwarded by a device (users specify this by providing a *flow filter* for each sketch). This is incorporated by summing u_h, u_m weighted by the probability that a sketch is updated for a particular packet. This probability is calculated based on the flow filter and traffic matrix to obtain the fraction of packets which satisfy

the flow filter. If this cannot be computed due to granularity of traffic matrix, we keep one additional counter per sketch that counts the number of packets that update the sketch.

Phase 3: Device Configuration. The Profiler must also build a model for how sketching and forwarding performance scales with device configuration (e.g. CPU cores, micro-engines). Since performance scaling may differ across bottlenecks, we study three sketch manifests: (1) Small sketches, which trigger compute bottlenecks; (2) Single large sketch, which trigger memory bottlenecks; and (3) No sketch, to study forwarding bottlenecks. Figure 4b shows these measurements for software switch, SoC smartNIC and FPGA NIC.

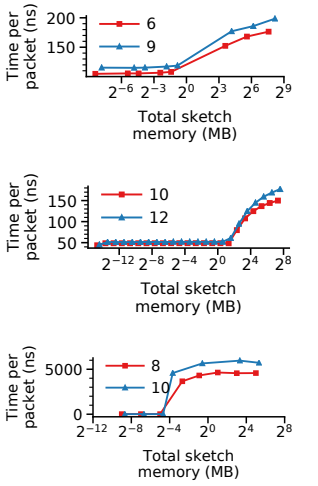
Based on these measurements, the Profiler estimates f, the fraction of parallelizable execution time by fitting Amdahl's law, and updates each of n_{fwd}, $n_{compute}$ and n_{memory} to include the effect of parallelism. For instance $n_{compute}$ (from Phase 1) becomes:

$$n_{compute} = (k_1 + u_h \cdot h)[(1 - f_c) + \frac{f_c}{c}]$$

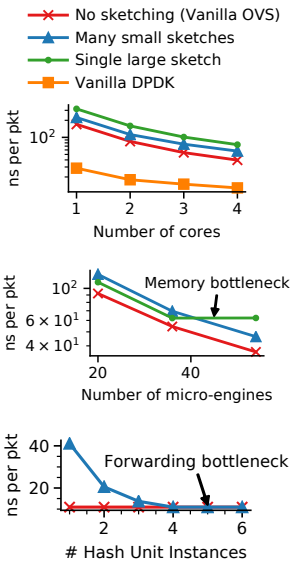
where f_c is fraction of parallelizable execution time when compute is bottleneck. We find that $f \approx 1$ for the software switch when any of forwarding, compute or memory is the bottleneck. For the SoC NIC, we find that f is ≈ 1 when forwarding or hashing is the bottleneck. However, when memory is the bottleneck, increasing micro-engines does not change packet rate, implying that f is 0 (Figure 4b). This is consistent with the fact that there is a single cache and DRAM (where sketch memory is allocated) – shared by all micro-engines – which becomes a bottleneck, as opposed to cores on a CPU which have their own caches, which allows for parallelism even when memory is the bottleneck. For the FPGA NIC, f is ≈ 1 when hashing is bottleneck otherwise it is 0. We measure the compute resources of FPGA in the units of a hash unit instance, each consuming 5 digital signal processors, 214 lookup tables, and 486 flip-flops.

Summary. The final relations encoding time per packet N in terms of number of operations u_h, u_m, sketch memory m and device parameters (h, $T(.)$, f's, constants) and device configuration are referred to as *device profiles*. This algebraic characterization of performance-resource trade-offs is used by the Optimizer for deciding sketch placement and resource allocation. In our current formulation, we don't explicitly model the impact of contention from non-monitoring tasks. We assume that compute resources are pinned to sketches and memory resources are explicitly allocated using technologies akin to Intel Cache Allocation Technology [54]. We also don't study the overhead of such isolation mechanisms.

Programmable switch [16] is a special case here as its resources are allocated by the compiler with guaranteed constant time per packet. Thus, the Profiler only needs to model the resources for different sketching manifests based on the resource usage output of the compiler.



(a) Phase 1 — Memory micro-benchmark. The different ridges correspond to cache levels.



(b) Phase 3

Figure 4: (a) Phase 1 — CPU with single core running OVS and sketching module (top), SoC NIC with 54 micro-engines (middle), FPGA with 10 hash instances (bottom). The numbers in the legend correspond to the number of sketch updates per packet.

(b) Phase 3 — Inverse throughput as CPU cores (top), SoC NIC micro-engines (middle) and FPGA hash unit instances (bottom) are varied. For FPGA NIC, single large sketch (not shown due to scale) is a flat line (memory bottleneck).

5 Optimizer

The goal of the Optimizer is to decide which sketch should be placed on which device and which resources each device should use while meeting the device constraints, monitoring requirements, traffic demands, and optimizing towards user-specified goals. We formulate the placement and resource allocation problem as a Mixed-Integer Bi-linear program (MIBLP), which is defined below with constants and variables described in Tables 3 and 4 respectively. While we investigate resource usage as an objective for concreteness, our formulation can be easily tweaked to handle other objective functions (Equation 7 in Appendix C).

Input: The input has the following three key features: (1) set of *devices* $\mathcal{D}$ in the network, along with their profiles generated using the Profiler (§4) and resource availability; (2) set of needed *sketches* $\mathcal{S}$, along with their configuration; (3) set of *Origin Destination (OD) pairs* $\mathcal{P}$.

In particular, each OD-pair is uniquely specified by: (1) device-level path in the network, (2) rate of traffic demand on that path, (3) set of sketches that should monitor traffic that is part of this OD-pair. With the OD-pair abstraction, we can handle the following cases:

- If there are multiple paths between an OD-pair, logically

$$\text{O1: resources} \quad \text{Minimize} \sum_{d \in \mathcal{D}} (res_d + mem_d), \quad \textbf{s.t.} \qquad (1)$$

$$\text{C1: coverage} \quad \sum_{d \in p_\pi} b_{(d,s)} \geq 1 \quad \forall p \in \mathcal{P}, \forall s \in p_s$$

$$\text{C2: accuracy} \quad mem_{(d,s)} \geq s_{mem} \cdot b_{(d,s)} \quad \forall s \in \mathcal{S}, \forall d \in \mathcal{D}$$

$$\text{C3: capacity} \quad \sum_{s \in \mathcal{S}} b_{(d,s)} \cdot s_{rows} \leq d_{rows}, \quad \text{and}$$
$$mem_d = \sum_{s \in \mathcal{S}} mem_{(d,s)} \leq d_{mem} \quad \forall d \in \mathcal{D}$$

$$\text{C4: profiles} \quad \forall d \in \mathcal{D}:$$
$$time_d = d_{time}(res_d, \mathcal{P}_d, \{(mem_{(d,s)}, b_{(d,s)}) | s \in \mathcal{S}\})$$

$$\text{C5: traffic} \quad time_d \leq \frac{1}{d_{\text{traffic}}} \quad \forall d \in \mathcal{D}, \quad \text{where}$$
$$d_{\text{traffic}} = \sum_{p \in \mathcal{P}_d} p_t, \quad \mathcal{P}_d = \{p | d \in p_\pi, p \in \mathcal{P}\}$$

Symbol	Interpretation
$\mathcal{D}, \mathcal{S}, \mathcal{P}$	Set of devices, sketches, and OD-pairs
p_s	Set of sketches for OD-pair p
p_π	Device level path for OD-pair p
p_t	Rate of traffic relevant for OD-pair p
s_{mem}, s_{rows}	Memory required for desired accuracy & number of rows for sketch s
d_{mem}	Maximum memory on device d
d_{rows}	Maximum rows that can fit on device d
d_{time}	(Function) representing the device profile (§4) for d (Time per packet in seconds)
$\mathcal{P}_d$	OD-pairs which pass through d
$d_{traffic}$	Total traffic rate in packet per second (pps) witness by device d

Table 3: Constants.

Domain	Symbol	Interpretation
$\mathbb{R}_{\geq 0}$	$mem_{(d,s)}$	Memory of sketch s on device d.
$\{0,1\}$	$b_{(d,s)}$	Is sketch s placed on device d
$\mathbb{Z}_{\geq 0}$	res_d	Resources allocated to device d, (e.g., processing cores, stages, functional units etc.)

Table 4: Variables.

distinct OD-pairs can be instantiated for each separate path.

- If part of the traffic in an OD-pair needs to be monitored by sketch A and other part using sketch B, then two logically distinct OD-pairs can be instantiated with the same path but specifying different sketches along with the appropriate rate of traffic which is relevant to each sketch.
- If the same query or sketch needs information about traffic on multiple OD-pairs, each OD-pair can refer to the same sketch identifier.
- Multiple sketches can monitor traffic in a single OD-pair. This would be used in the cases when we need to maintain a statistic for different dimensions of the same traffic (e.g., dimension 1: distribution of DstIPs for each source and dimension 2: distribution of SrcIPs for each destination).

This input is compiled from the high-level measurement requirements specified by the user. The traffic demands (packet rates) are estimated using the traffic matrix and the paths are obtained using the routing information and flow filters specified for each sketch by the user.

Outputs and constraints: The Optimizer decides which sketch should be placed on which device. This is indicated through variables $b_{(d,s)}$. While doing so, the Optimizer ensures that for each OD-pair, each sketch of that OD-pair is placed on at least one of the devices lying on the OD-pair's path (C1: flow coverage in Equation 1). The memory for each sketch is directly determined by the accuracy required for that sketch (C2: monitoring accuracy). Each device is constrained by memory capacity and some devices may have constraints on row capacity (e.g., due to limited stages in programmable switch) (C3: device capacity). C4 (device profiles) encodes the relationship between time per packet, the sketch parameters, device parameters and the device configuration as described in Section 4. The processing overhead on each device should be such that the overhead does not stall the traffic flowing through the device (C5: forwarding performance). Note that C4 is natively expressed through Gurobi's API using piece-wise linear [55] and bi-linear constraints [56]. We elaborate on this in Appendix C.

Measurement Accuracy: Different feasible solutions may deploy sketches at different locations in a network (e.g., Figure 1) and even create multiple instances of the same sketch. We note that our sketch placements make no impact on the monitoring accuracy. This is because, these multiple instances are linearly merged (§2) at the central controller. The merge is possible as instances of the same sketch share the same hash functions, memory configuration. The merge does not lead to over/under counting as we ensure that each packet updates exactly one instance of all the required sketches. This is ensured as: (1) constraint C1 guarantees that there is at least one instance of the required sketches on the path of each OD-pair, and (2) we generate sketch manifests so that exactly one of these instances is chosen (arbitrarily) to be updated for each OD-pair.

6 Scalability and Dynamics

Solving the MI-BLP in Gurobi can take more than a few hours even for modestly sized data center topologies with thousands of devices. For quick responses to network dynamics and scaling to more devices, we use a three step approach:

- *Step 1*: Partition the network topology into disjoint clusters
- *Step 2*: Run Optimizer to assign sketches to the clusters[2]
- *Step 3*: For each cluster, run Optimizer to place sketches onto devices *within* the cluster.

Since the placement decision for each cluster is done independently, Optimizer instances can be spawned in parallel,

[2]For this step, the traffic demands and device profiles (C4-5 in equation 1) are not used, as the Profiler does not model the performance of clusters.

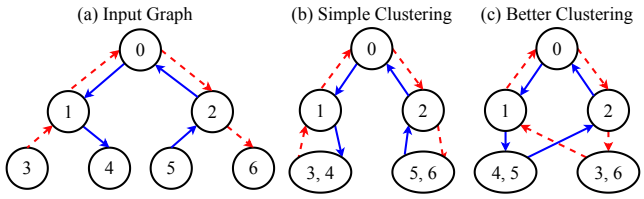


Figure 5: Different ways to cluster a graph.

which makes this approach scalable. Note that, Step 2 itself can consume significant time if the number of clusters is large. We address this by recursively applying Step 1 and 2 to build a hierarchy of clusters. We determine the threshold to apply the recursive step by modeling the cluster size-compute time relationship (Figure 14 in Appendix A) and choosing the largest cluster size with an acceptable run-time. We also add a *fast path* for quick response to cater to network dynamics that *directly affect* only a small subset of devices.

6.1 Clustering Approach

Partitioning the network devices into clusters will inevitably hide some details about the topology and create a trade-off between optimality and solving time. This is because when the Optimizer is run to place sketches onto the cluster (Step 2), it may choose a sub-optimal (or even infeasible) cluster as it does not know what is inside the cluster. Ideally, we want to cluster the topology in a way that significantly accelerates the solving process while incurring minimal optimality loss.

Clustering Examples. We observe that naïvely clustering the topology graph based on the hierarchy of the topology or applying graph clustering algorithms (e.g., spectral clustering [51]) can lead to sub-optimal or even infeasible sub-problems. Figure 5 illustrates this in an example topology of 7 devices. In this example, the edges show the paths of the OD-pairs and the colors (or line-styles) of the edges correspond to different OD-pairs. We need to deploy 2 sketches, each of which monitors one of the OD-pairs shown in blue (solid) and red (dotted).

Simple clustering: By directly applying spectral clustering on network topology, we obtain the result in Figure 5(b). Based on this clustering, assume that in *Step 2*, the Optimizer decided to place one sketch on cluster (3,4) and the other sketch on cluster (5,6). Further assume that the sketch placed on cluster (3,4) monitored the red OD-pair. When the Optimizer again runs *Step 3* for devices within cluster (3,4), since only device 3 sees packets on the red path, the sketch for the red OD-pair can only be placed on device 3. If device 3 is currently not available to place the sketch or device 6 is in fact a better allocation, the simple clustering will lead to an infeasible or sub-optimal solution.

Better clustering: If we cluster as Figure 5(c), the sketch for the red (dotted) OD-pair could be assigned to cluster (3,6) and the Optimizer running within that cluster retains its freedom to place the sketch on device 3 or 6.

Our Design. We learned from the above example that we should keep nodes that *communicate* with each other in the same cluster, where communicate means that there is an OD-pair that has a path connecting the nodes. This should be done irrespective of the number of network-level hops (physical or logical links) between them. This provides the Optimizer in *Step 3* additional placement choices for sketches within the cluster sub-problem.

We can incorporate communication affinity by applying spectral clustering on the communication graph (G_c), where vertices are network devices, with edges between all pairs of devices which communicate with each other. While this approach works for general network environments, we find that spectral clustering itself is time consuming for large networks [57]. Thus, we imitate spectral clustering using a domain-specific heuristic and are investigating faster alternatives and implementations of spectral clustering.

Our heuristic is based on the observation that many clustering solutions preserve enough flexibility for sketch placement yielding good performance (Figure 12 in Appendix A). This observation was made when exploring the space of possible clustering solutions and their impact on MIP objective using simulated annealing [58]. Our heuristic works in a multi-tenant setting where tenants share the network but not the end-hosts. For building clusters, we instantiate a cluster for end-hosts of each tenant. Then, to ensure clusters are evenly sized, we arbitrarily merge (or split) the clusters if they are too small (or big). Finally, the switches and NICs are assigned to the cluster of the end host with which they have highest affinity. In Appendix A, we discuss how to choose cluster sizes and provide examples of clustering output for different clustering techniques.

6.2 Fast Path

The Fast Path further improves response time for network dynamics including: (1) Topology change in the path (e.g., VM migration); (2) Monitoring query (sketch) change from the user/operator; (3) Traffic change in the OD-pair (e.g., traffic demand variations); (4) Available resource change due to the dynamics of other non-monitoring services running on the shared devices. (1)–(3) reflect as changes in OD-pairs ($\mathcal{P}$) and (4) reflects as changes in devices ($\mathcal{D}$).

The key observation we have from §6.1 is that sub-problems should preserve enough placement choices for sketches. Based on that, we find that, on network change events, recomputing placement only for the set of devices $\mathcal{A}$ *directly affected* by the changes is sufficient. We compute this set through the following process: (a) For each changed device $d \in \mathcal{D}$, we add d to $\mathcal{A}$ and, for each sketch s currently placed on d, we add the OD-pair(s) monitored by s to the set of changed OD-pairs. (b) For each changed OD-pair (due to previous step or otherwise), we add all devices specified in the path of the OD-pair to $\mathcal{A}$. The Optimizer is then run

only for the devices in set $\mathcal{A}$ and the sketches specified in the changed OD-pairs, including the sketches already mapped to devices in $\mathcal{A}$.

7 Evaluation and Implementation

We implement HeteroSketch and evaluate its effectiveness. Our major findings are as follows:

- HeteroSketch's Profiler accurately characterizes the performance of devices across a variety of sketch manifests and device configurations (§7.1)
- HeteroSketch's Optimizer is able to (1) reduce resource footprints and obtain feasible solutions when prior approaches fail, and (2) scale to large topologies (> 40,000 devices) while preserving good quality solutions. (§7.2)
- HeteroSketch's Fast Path allows prompt responses to network dynamics including changes in topology, traffic, query, and available resources. (§7.3)

Implementation. For the software switch, the sketching modules are implemented as a part of the OVS data plane. For the SmartNIC, we use the internal and external memory regions for storing sketch state as these are accessible from all the micro-engines unlike local and island-specific memory regions. The Optimizer is run on an Intel® Xeon® CPU E5-2680 v2 processor @ 2.80GHz with 128 GB RAM.

7.1 Performance Profiler

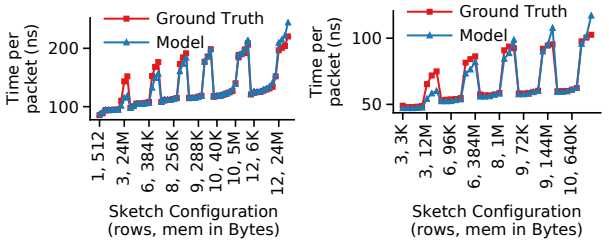
Setup. This evaluation uses the same three device setup introduced in §4 which was used to create the device profiles (Table 2). We use `dpdk-pktgen` [59] to generate traffic and configure `dpdk-testpmd` [60] to measure receive rate. The sender generates min-sized (64 Byte) packets to measure the maximum packets per second that can be processed. We use source IPs as the flow keys for the sketches, which are taken from a uniform random distribution. This is done in order to use the Optimizer to allocate resources for worst-case (uniform) traffic scenarios. We discuss in Appendix B, how other traffic distributions could be accommodated.

Workloads. For generating sketch manifests, we consider a range of configurations for three different types of sketches: Count-Min, Count Sketch and UnivMon. For each sketch, we vary the number of rows from 1 to 12 and, for memory, we vary the counters per row from 1 to 2^{22} ($\approx$ 4 million) in steps of powers of 2. For UnivMon, we vary levels from 2^2 to 2^5 in steps of powers of 2. For generating device configurations, we vary the SoC NIC micro-engines from 20 to 54, for the software switch, vary cores from 1 to 4, and for the FPGA NIC, vary maximum allowed hash instances from 1 to 12. Large number of rows emulate multiple sketches per device.

Results. Table 5 summarizes the results of the experiments and Figure 6 shows results for a subset of the experiments.

Sketch	CPU	SoC NIC	FPGA
Count-Min Sketch	3.08, 9.63	3.68, 12.99	1.9, 4.1
Count Sketch	5.61, 8.51	1.26, 4.51	2.2, 4.14
UnivMon	2.80, 6.38	2.38, 3.75	2.28, 5.88

Table 5: Profiler Evaluation — Each sketch–device combination reports the (mean, 90^{th} percentile) of percent error $\left(\left|\frac{actual-model}{actual}\right| * 100\right)$ for the time per packet metric.



(a) 2 Cores **(b) 4 Cores**

Figure 6: Performance Profiler — CPU model evaluation for Count-Min Sketch.

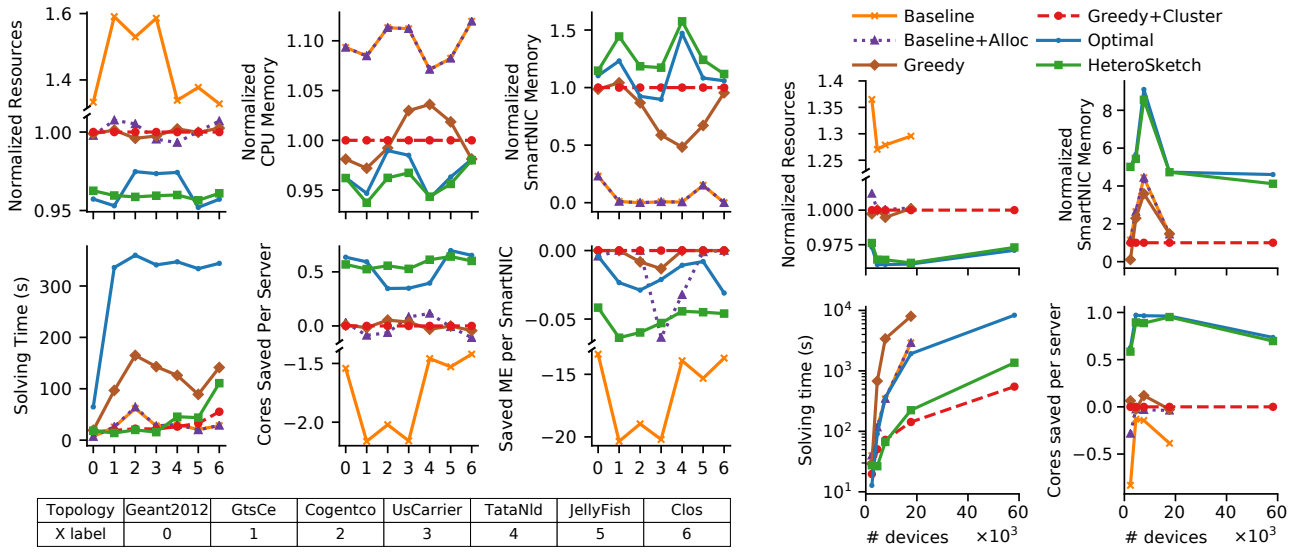
Figures for additional experiments can be found in Appendix B. These figures compare the time per packet estimated by the Profiler's model and the ground truth. Over all the combinations of sketch manifests and device configurations, the Profiler's model is within 5.61% of the ground truth on average and within 13% in the tail cases. We don't show results for programmable switch as it guarantees line rate if the program fits (which is captured by the capacity constraints in the Optimizer). We observe that most profiling errors occur in larger sketch configurations, when off-chip memory is used. Based on our estimates, such errors would disturb the resource allocation for less than 5% devices when profiles are off by 10%. We discuss the impact of profiling errors on the Optimizer in more detail in Appendix B.

7.2 Optimizer

We evaluate the solutions generated by the Optimizer on two key metrics: (1) resource-efficiency benefits and (2) the optimization run time. We use the device profiles to estimate the resource usage and performance for the sketch placements generated by different optimization schemes. We use the following methodology to generate input scenarios:

Topologies. We conduct two studies: (a) *Topology* – variation with topology structure. We use selection of topologies from the Internet Topology Zoo [61], JellyFish [62] and a three-level Fat-Tree data center topology (Clos) [63, 64] (Figure 7a). (b) *Scale* – variation with topology size. We use a Clos topology with varied degrees (number of *pods*) from 16 to 48 (Table 6, Figure 7b). We extend the Points-of-Presence (POP) topologies [61] to include servers and NICs based on principles from [65].

For both studies, we focus on a multi-tenant setting, where different tenants submit monitoring tasks to a central system (e.g., the cloud or Internet service provider). This allows us



(a) Topology Study — The Clos topology used has 20 pods, JellyFish has the same number of devices as the Clos topology (2000 servers/NICs each and 500 switches). Note, a time limit of 300s for the MIP solver was used.

(b) Scale Study — These experiments were performed on inputs described in Table 6.

Figure 7: Optimizer Evaluation — Compute resources are shown in terms of amount saved relative to Greedy+Cluster (negative compute resources implies more resource consumption than Greedy+Cluster). Total resources and memory resources are normalized w.r.t Greedy+Cluster. Both sub-figures share the same legend. Additional details in Appendix C (Figures 20 & 21).

Fat-tree Degree	Sketch load (Y)	Servers / NICs	Switches	Total devices
16	1	1,024	320	2,368
20	3	2,000	500	4,500
24	3	3,456	720	7,632
32	4	8,192	1,280	17,664
48	4	27,648	2,880	58,176

Table 6: Topologies and Workloads.

to stress test the schemes under a diverse set of monitoring requirements. We assign X servers to each tenant where X is taken from the uniform distribution $\mathcal{U}(6,12)$. We set the capacity of each server link to 25 Mpps (64B packets). This is equivalent to the throughput of vanilla OVS with 4 cores. We randomly assign half of the servers with the SoC SmartNIC and other half with the FPGA NIC.

Monitoring requirements. The total number of queries (sketches) is set equal to Y times the number of servers in the network. These monitoring tasks are evenly partitioned among the tenants. We vary the sketching load (Y) between 1 (low) and 4 (high). Low load is used to study the system when each device in the network runs much below the total monitoring capacity they can handle. High load is used to study the system under stress. Table 6 shows the load used for different topologies. The monitoring tasks are equally divided between Count-Min, Count Sketch and UnivMon sketches.

OD-pairs. Each tenant specifies M OD-pairs from the set of servers assigned to them, where $M \sim \mathcal{U}(64,96)$ and each OD-pair is monitored by K randomly chosen sketches of the tenant where $K \sim \mathcal{U}(1,3)$. Since we don't have access to the monitoring demands from different operators, we select OD-pairs, routes (paths) and traffic demands iteratively to ensure: (1) traffic is evenly distributed between OD-pairs, and (2) link utilization is at least 90% to stress the system.

Compared Schemes. As shown in Figure 7b, we compare HeteroSketch (6) against five other schemes (1)–(5):

(1) Baseline: Capacity-aware placement with *static* resource allocation, i.e., placing sketches to minimize the sketch memory with compute resource assigned apriori to cores=5, micro-engines=54 (equal to resources exposed to Optimizer). This is closest to UnivMon [11].

(2) Baseline+Alloc: The placement of sketches is done in the same manner as in Baseline. Instead of static resource assignment, just enough resources are allocated to meet the traffic and sketching demands based on the device profiles. This is used to investigate benefits obtained solely from profiling-aware resource allocation.

(3) Greedy: This is a strawman extension to the baseline which prioritizes placing sketches on programmable switches over CPUs and SmartNICs because of their line-rate guarantees. Resource allocation is done using device profiles similar to Baseline+Alloc. Prioritizing sketch deployment on switches is a reasonable heuristic when sketch load (Y) is low (first data point of Figure 7b, gap between resource usage for Baseline+Alloc and Greedy).

(4) Greedy+Cluster: To compare the optimality of our scheme to prior work for larger topologies, we extend the Greedy strategy to use our clustering optimization.

(5) HeteroSketch w/o clustering (Optimal): Joint placement and resource allocation using the formulation in Equation 1

Sketch	CPU	SoC NIC	FPGA	Switch
Count-Min	4112	2001	2111	1563
Count	4172	2066	2106	232
UnivMon	4169	2120	2049	143

Table 7: Sources of benefits (Sketch-Device Affinity) — Total number of sketches on each device. (Clos $pods = 16, Y = 4$.)

without the clustering approach.

(6) HeteroSketch: Joint placement and resource allocation with the clustering optimization.

Results. In Figure 7, we can see that HeteroSketch is able to lower resource utilization significantly (close to optimal) and saves between half to one CPU core per server on average compared to Greedy+Cluster ($20-30\%$ improvement). Compared to Baseline, HeteroSketch saves around 2.5 cores per server and about $15-20$ micro-engines per SoC NIC ($40-60\%$ improvement). It accomplishes this without incurring significant time to compute the placement of measurement tasks even as topologies scale to more than 40k devices. We compute total resources (HeteroSketch MIP objective) as a weighted sum of all devices resources including CPU cores, micro-engines, FPGA hash unit instances and memory, and normalize it by the total resources used by Greedy+Cluster. This is shown as *normalized resources* in Figure 7.

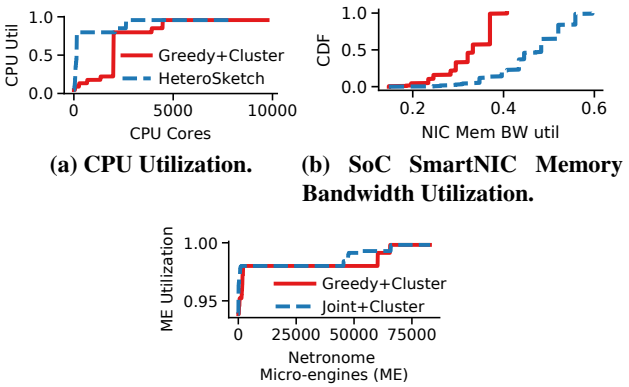
Sources of Benefits. We explore the benefits obtained from different features of HeteroSketch.

Bottleneck awareness: The device profiles are successful in incorporating capabilities of different device architectures, this allows HeteroSketch to allocate just enough resources to meet the sketching and forwarding demands. The gap in normalized resources between Baseline and Baseline+Alloc in Figures 7a and 7b demonstrates benefits attained solely from profiling aware resource allocation. For instance, on the SoC SmartNIC, when memory is the bottleneck, more micro-engines do not improve forwarding performance (Figure 4b).

Efficient use of resources: We see from Figure 8a that HeteroSketch allocates 20% fewer CPU cores (8k vs 10k cores) but uses the cores it allocates more effectively (each core is >80% utilized). Specifically, on the software switch, the cores are configured to poll NICs for packets (for performance reasons), as a result, CPU cycles are wasted busy polling when there are no packets in the NIC buffers. With the help of device profiles, the Optimizer is able to consolidate load towards cores which would otherwise waste cycles. Similar trends are observed for other resources including SoC memory bandwidth (Figure 8b), and SoC micro-engines (Figure 8c).

Ability to trade-off resources: For the scale study (Figure 7b), we set lower weightage to SoC SmartNIC memory relative to the topology study (Figure 7a). We observe in Figure 7b that HeteroSketch is able to incorporate this by saving more number of cores per server at the cost of SoC SmartNIC memory usage.

Sketch-Device affinity: We show the number of sketches instantiated of each type on each device in Table 7. Recall that



(a) CPU Utilization. **(b) SoC SmartNIC Memory Bandwidth Utilization.**

(c) SoC SmartNIC Micro-engine Utilization.

Figure 8: Sources of Benefits (Efficient use of resources) — (Clos $pods = 20$).

the monitoring requirement specified equal number (≈ 1360) of queries for each sketch type. Multiple instances of each sketch are created to meet the coverage requirements. We make three key observations here: (1) HeteroSketch tries to place heavier sketches (e.g. UnivMon) on better vantage points so as to reduce the total required instances. (2) The switch statically allocates resources to each sketch while other devices can share resources across sketches. Due to this HeteroSketch places less number of sketches on the switch, especially for Count Sketch and UnivMon due to more number of operations. (3) HeteroSketch instantiates relatively more number of UnivMon sketches on CPU and SoC SmartNIC as they have relatively larger and faster memories.

Clustering Algorithm. Figure 7b and 7a also suggest that our clustering technique does not significantly degrade resource efficiency. Clustering imposes a trade-off between optimality and solving time which we explore in more detail in Appendix A. We see in Figure 7a, that HeteroSketch finds better solutions than the Optimal scheme when configured with a time limit, achieving a better trade-off between optimality and solving time than the MIP solver.

For evaluating the Optimizer, we used the multi-tenant clustering heuristic developed in §6.1. We investigated use of other algorithms to cluster the communication graph (§6.1) including KMedoids, HDBSCAN, modularity maximization [57, 66]. Unfortunately, these techniques yield infeasible sub-problems in Step 3 of §6.1 for the inputs that we used.

Note, while it is true that our assumption about one server being solely used by one tenant makes the optimization problem instance easier, despite that, the MIP solver still needs explicit guidance in the form of clustering for speed up. This can be seen from difference in solving time with and without clustering in Figure 7.

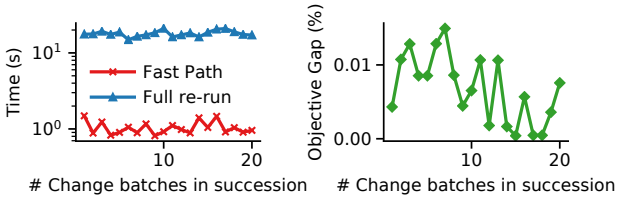


Figure 9: Dynamics — variation with number of changes in succession (Clos topology with 16 pods).

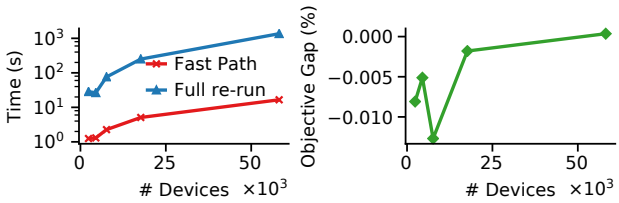


Figure 10: Dynamics — variation with topology size for a single change. Negative objective gap means that Fast Path consumes less resources than full re-run. This can happen because both use the clustering heuristic which does not guarantee strictly optimal solutions. (Conducted on topologies in Table 6.)

7.3 Dynamics - Fast Path

As described in §6, changes in OD-pairs can correspond to changes in traffic, monitoring requirement, and topology. To generate such changes, we generate inputs for the Optimizer as described in §7.2. Then, we randomly keep $10n$ of the generated OD-pairs aside to serve as change batches, where n is the number of batches and each batch has 10 changed OD-pairs. We also generate resource availability changes in each batch by randomly sampling 10 devices and randomly increasing/decreasing their resource availability (e.g., micro-engines/memory) by 20%. We find that changes in a batch amount to roughly 50 to 150 devices being directly affected ($|\mathcal{A}|$ in §6.2) across the topologies in Table 6. In general, $|\mathcal{A}|$ can depend on the size of the topology and monitoring requirements. For the Clos topology, since path lengths are equal irrespective of number of pods, we see little dependence of pod count on $|\mathcal{A}|$.

Despite being run only for the subset of affected devices, successive runs of the Optimizer don't diverge from the global solution generated by re-running the Optimizer over the entire topology using clustering (Figure 9). Further, the response time to cater to dynamics is reasonably low with the Fast Path even as we scale the topology size (Figure 10). The Fast Path uses the clustering optimization when the set of affected devices is larger than the optimal clustering size and can amount to a full run of the Optimizer in the worst case. Fast Path, together with the clustering heuristic enables a response time of a few seconds (the set of affected devices is small) to a few minutes (when run over entire network). Note, within the scope of this work, we don't study how the new placement can be configured consistently and quickly. These are active areas of research [35, 67].

8 Other Related Work

Other related work not covered in §2 can be classified into four categories:

Sketch resource allocation. SCREAM [68] allocates memory for a sketch based on temporal and spatial changes in traffic moments for a fixed sketch placement. Open Sketch [10] optimally selects sketch algorithm and configuration for a given query. Both are complementary to our work.

Sketch implementations for different hardware. Our work relies on state-of-the-art implementations of sketches for different hardware. Fortunately, recent efforts [12, 13, 22] have focused on addressing the bottlenecks of sketching algorithms in software switches and have demonstrated efficient implementations in programmable switches or NICs [11, 12, 31].

Other work in network monitoring. HeteroSketch's goal is to support network-wide flow monitoring. Numerous complementary efforts focus on either fine-grained and adaptive flow monitoring (e.g., [36, 38, 69]), diagnosis (e.g., [70–72]), or network performance-related objectives (e.g., [73, 74]).

Efforts in speeding up network-wide optimizations. Concurrent with our work, Abuzaid et al. in [75] explore the use of clustering to speed up network flow problems. While our work tries to maintain feasiblity of sub-problems through carefully deciding which devices to cluster together, they impose additional constraints while conducting flow allocations over clusters to ensure that the subsequent sub-problems are feasible. We leave exploration of such a technique in the context of sketch placement for future work.

9 Conclusions

We observe that existing efforts on sketch-based network-wide monitoring remain impractical as they fail to cope with the key requirements of heterogeneity and dynamics in the network. We propose HeteroSketch as a coordinated solution to achieve optimized task placement and resource allocation over heterogeneous networks. HeteroSketch precisely characterizes the performance of sketches on diverse devices and is integrated with a clustering technique to handle networks scale and dynamics. Our evaluation demonstrates that HeteroSketch scales to toplogies with tens of thousands devices with near optimal resource efficiency. We posit that our system can more generally be applied to allocate resources for other networked applications in heterogeneous networks, and we plan to explore this for future work.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. We would like to thank Hun Namkung, Pouya Haghi, Anqi Guo, Zhipeng Zhao, and Nirav Atre for assistance with hardware implementations. This work is supported in part by NSF Grants CNS-1700521, CNS-2106946, CNS-2107086, SaTC-2132643.

References

[1] Mohammad Alizadeh et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *Proceedings of the 2014 ACM conference on SIGCOMM*. 2014, pp. 503–514.

[2] Mohammad Alizadeh et al. "PFabric: Minimal near-Optimal Datacenter Transport". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 435–446.

[3] Theophilus Benson et al. "MicroTE: Fine grained traffic engineering for data centers". In: *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. 2011, pp. 1–12.

[4] James McCauley et al. "Thoughts on Load Distribution and the Role of Programmable Switches". In: *ACM SIGCOMM Computer Communication Review* 49.1 (2019), pp. 18–23.

[5] Rui Miao et al. "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.

[6] Pedro Garcia-Teodoro et al. "Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges". In: *computers & security* 28.1-2 (2009), pp. 18–28.

[7] L. Ying, R. Srikant, and X. Kang. "The power of slightly more than one sample in randomized load balancing". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. 2015, pp. 1131–1139.

[8] Graham Cormode. "Count-Min Sketch". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 1–6.

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2380 LNCS. 2002, pp. 693–703.

[10] Minlan Yu, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch". In: *Nsdi '13*. 2013, pp. 29–42.

[11] Zaoxing Liu et al. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 101–114.

[12] Tong Yang et al. "Elastic sketch: Adaptive and fast network-wide measurements". In: *SIGCOMM 2018 - Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 561–575.

[13] Zaoxing Liu et al. "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.

[14] B. Claise. *Cisco systems NetFlow services export version 9*. RFC 3954.

[15] Peter Phaal, Sonia Panchen, and Neil McKee. "InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks". In: (2001).

[16] *Tofino 2 | Barefoot*. URL: https://www.barefootnetworks.com/products/brief-tofino-2 (visited on 04/28/2020).

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

[18] *Netronome*. URL: https://www.netronome.com/products/smartnic/overview (visited on 04/28/2020).

[19] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66.

[20] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130.

[21] *BESS: Berkeley Extensible Software Switch*. URL: http://span.cs.berkeley.edu/bess.html (visited on 09/12/2020).

[22] Qun Huang et al. "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.

[23] Vyas Sekar et al. "CSAMP: A System for Network-Wide Flow Monitoring". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, pp. 233–246.

[24] Xin Jin et al. "Netchain: Scale-free sub-rtt coordination". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 35–49.

[25] Xin Jin et al. “Netcache: Balancing key-value stores with fast in-network caching”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 121–136.

[26] Rui Miao et al. “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*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 15–28.

[27] *Alveo U280 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html (visited on 03/05/2021).

[28] Antonis Manousis et al. “Contention-Aware Performance Prediction For Virtualized Network Functions”. 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*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 270–282.

[29] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. “Toward Predictable Performance in Software Packet-Processing Platforms”. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 141–154.

[30] *Gurobi - The fastest solver - Gurobi*. URL: https://www.gurobi.com (visited on 04/29/2020).

[31] Vibhaalakshmi Sivaraman et al. “Heavy-hitter detection entirely in the data plane”. In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 164–176.

[32] Pat Bosshart et al. “Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM ’13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 99–110.

[33] Yin Zhang et al. “On the characteristics and origins of internet flow rates”. In: *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 2002, pp. 309–322.

[34] Masoud Moshref et al. “Scalable Rule Management for Data Centers”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 157–170.

[35] Qun Huang et al. “OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy”. 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*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 404–421.

[36] Arpit Gupta et al. “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.

[37] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. “ProgME: towards programmable network measurement”. In: *IEEE/ACM Transactions on Networking* 19.1 (2010), pp. 115–128.

[38] Rob Harrison et al. “Network-wide heavy hitter detection with commodity switches”. In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.

[39] Masoud Moshref et al. “DREAM: Dynamic Resource Allocation for Software-Defined Measurement”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 419–430.

[40] Nick Duffield, Carsten Lund, and Mikkel Thorup. “Estimating flow distributions from sampled flow statistics”. In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 2003, pp. 325–336.

[41] Cristian Estan and George Varghese. “New directions in traffic measurement and accounting”. In: *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 2002, pp. 323–336.

[42] Noga Alon, Yossi Matias, and Mario Szegedy. “The space complexity of approximating the frequency moments”. In: *Journal of Computer and system sciences* 58.1 (1999), pp. 137–147.

[43] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “SpaceSaving: Efficient Computation of Frequent and Top-k Elements in Data Streams”. In: *Proceedings of the 10th international conference on Database Theory*. 2005, pp. 398–412.

[44] Balachander Krishnamurthy et al. “Sketch-based change detection: methods, evaluation, and applications”. In: *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 2003, pp. 234–247.

[45] George Nychis et al. "An empirical evaluation of entropy-based traffic anomaly detection". In: *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. 2008, pp. 151–156.

[46] Peter Clifford and Ioana Ada Cosma. "A simple sketching algorithm for entropy estimation". In: *arXiv preprint arXiv:0908.3961* (2009).

[47] Ashwin Lall et al. "Data streaming algorithms for estimating entropy of network traffic". In: *ACM SIGMETRICS Performance Evaluation Review* 34.1 (2006), pp. 145–156.

[48] Pankaj K Agarwal et al. "Mergeable summaries". In: *ACM TODS* (2013).

[49] Behnaz Arzani et al. "PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 797–815.

[50] Changhoon Kim et al. "In-band Network Telemetry via Programmable Dataplanes". In: *Deme session of ACM SIGCOMM*. 2015.

[51] Ulrike von Luxburg. *A Tutorial on Spectral Clustering*. 2007. arXiv: 0711.0189 [cs.DS].

[52] Ran Ben Basat et al. "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.

[53] *ConnectX® Ethernet Adapters*. URL: https://www.mellanox.com/products/ethernet/connectx-smartnic (visited on 04/28/2020).

[54] *Introduction to Cache Allocation Technology in the Intel® Xeon®...* URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html (visited on 09/17/2020).

[55] *Constraints*. URL: https://www.gurobi.com/documentation/9.1/refman/constraints.html#subsubsection:GenConstrSimple (visited on 03/05/2021).

[56] *Non-Convex Quadratic Optimization - Gurobi*. URL: https://www.gurobi.com/resource/non-convex-quadratic-optimization (visited on 03/05/2021).

[57] *Benchmarking Performance and Scaling of Python Clustering Algorithms — hdbscan 0.8.1 documentation*. URL: https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html (visited on 09/16/2020).

[58] Dimitris Bertsimas, John Tsitsiklis, et al. "Simulated Annealing". In: *Statistical science* 8.1 (1993), pp. 10–15.

[59] *The Pktgen Application — Pktgen 3.2.4 documentation*. URL: https://pktgen-dpdk.readthedocs.io/en/latest (visited on 05/01/2020).

[60] *Testpmd Application User Guide — Data Plane Development Kit 20.05.0 documentation*. URL: https://doc.dpdk.org/guides/testpmd_app_ug (visited on 05/26/2020).

[61] S. Knight et al. "The Internet Topology Zoo". In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775.

[62] Ankit Singla et al. "Jellyfish: Networking data centers randomly". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 225–238.

[63] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 63–74.

[64] frenetic-lang. *ocaml-topology*. URL: https://github.com/frenetic-lang/ocaml-topology (visited on 05/26/2020).

[65] Lun Li et al. "A First-Principles Approach To Understanding the Internet's Router-Level Topology". In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 3–14.

[66] *Structure & Strangeness*. URL: https://www.cs.unm.edu/~aaron/research/fastmodularity.htm (visited on 03/05/2021).

[67] Xiaoqi Chen et al. "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*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 226–239.

[68] Masoud Moshref et al. "SCREAM: Sketch Resource Allocation for Software-Defined Measurement". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '15. Heidelberg, Germany: Association for Computing Machinery, 2015.

[69] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. “Distributed network monitoring and debugging with switchpointer”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 453–456.

[70] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. “Confluo: Distributed monitoring and diagnosis stack for high-speed networks”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 421–436.

[71] Thomas Holterbach et al. “Blink: Fast connectivity recovery entirely in the data plane”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 161–176.

[72] Chang Lou, Peng Huang, and Scott Smith. “Understanding, Detecting and Localizing Partial Failures in Large System Software”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 559–574.

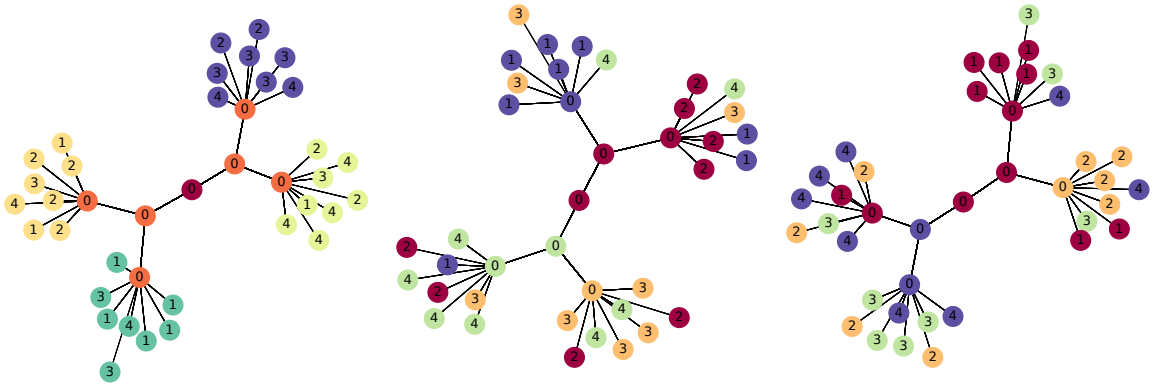
[73] Wei Le and Mary Lou Soffa. “Marple: a demand-driven path-sensitive buffer overflow detector”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 272–282.

[74] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. “Dapper: Data plane performance diagnosis of tcp”. In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 61–74.

[75] Firas Abuzaid et al. “Contracting Wide-area Network Topologies to Solve Flow Problems Quickly”. In: *NSDI*. USENIX. Nov. 2020.

[76] Mingran Yang et al. “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*. MobiCom ’20. London, United Kingdom: Association for Computing Machinery, 2020.

[77] *Working With Multiple Objective*. URL: https://www.gurobi.com/documentation/9.0/refman/working_with_multiple_obje.html (visited on 09/17/2020).



(a) Spectral Clustering on Topology

(b) Spectral Clustering on Communication Graph

(c) Clustering using multi-tenant heuristic

Figure 11: Output of clustering

A Clustering Details

Clustering Heuristic. We performed simulated annealing to explore the space of clustering solutions. Figure 12 shows the annealing in action. We find that there are many clustering solutions that result in optimization solutions which are close to optimal.

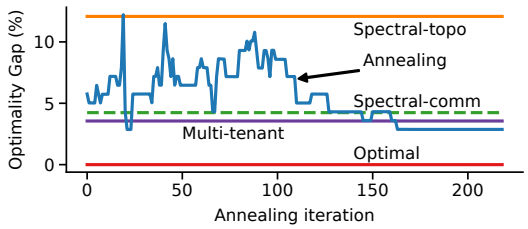


Figure 12: Simulated Annealing — Spectral-topo refers to spectral clustering over network topology. Similarly, Spectral-comm is over the communication graph. Multi-tenant refers to our domain specific heuristic. Optimal refers to no clustering.

Clustering Output. Figure 11 illustrates examples of the clustering output for different clustering techniques. The topology shown is a simple tree topology with only servers and switches (without NICs) for ease of visualization. The nodes marked '0' are switches and the other nodes are servers. The non-zero numbers on the servers signify that servers with the same number are communicating with each other. The colors signify that devices having the same color are in the same cluster.

Optimality vs. Solving Time Trade-off. Figure 7b and 7a also suggest that our clustering technique does not significantly degrade resource efficiency. This is counter-intuitive since clusters limit the types of optimization possible. We explore this in in Figure 13. Cluster size represents a trade-off between optimality and solving time, i.e. smaller clusters help reduce solving time at the cost of optimality. We observe that the even for relatively small clusters (20 devices), the optimality gap is very low (< 0.4%), allowing us to choose small clusters to reduce the run-time while preserving close to optimal solutions. The extreme case of all devices within the same cluster mimics the case of no clustering. In this case, all sketches would be assigned to the only available cluster and then the Optimizer is run to place sketches on the devices within that single cluster, effectively deciding between all devices of the topology.

The MIP solver also natively allows trading-off solving time for optimality through configuration of a time limit. In Figure 7a, we configured a time limit of 300s. We find that HeteroSketch is able to produce better quality solutions in lesser time, achieving a better trade-off between optimality and solving time.

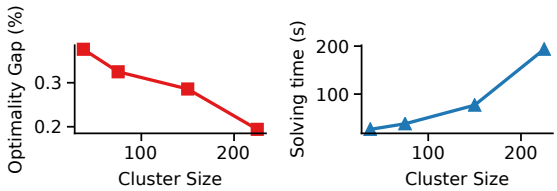


Figure 13: Cluster size — optimality gap vs runtime. The optimality gap is the gap between the objective of HeteroSketch relative to the objective value of Optimal. (For experiment with $pods = 24$).

Choosing Cluster Size. As we see in Figure 13, the clustering heuristic provides a trade-off between solution quality (optimality) and optimization run-time. We want to be able to select the largest cluster size which allows an acceptable run-time. To do this, we look at the knee of the graph between solving time and number of devices (Figure 14). Since, the solving time depends not only on the network topology but also on the monitoring load (Y, see §7.2), we need to recompute this graph whenever the monitoring load changes significantly. We use the following procedure to quickly re-

compute the solving time vs number of devices graph: we divide the network topology into clusters of different sizes. Then we run the Optimizer for a sample of these clusters which have different sizes, and we effectively obtain solving time as a function of number of nodes (cluster size). Figure 14 shows this in action.

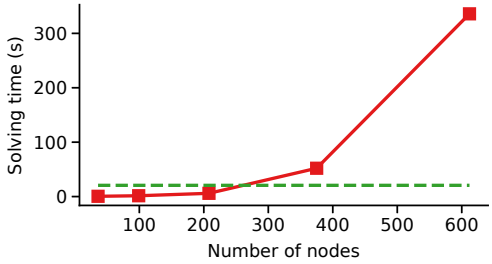


Figure 14: Determining cluster sizes — X-axis represents the number of nodes in a cluster, and Y-axis denotes maximum time to solve *just one* cluster (averaged over 5 clusters of the same size in the topology)

B Performance Profiler Details

Non-uniform Memory Access Pattern. Sketches such as UnivMon which have multiple control paths can result in non-uniform access of the working memory set even if the traffic is uniform. There are at least the following two ways to handle such cases: (1) estimating *effective memory size* that is accessed uniformly, (2) estimating *hit rates* to different levels of memory hierarchy. In what follows provide some background on the operations of the UnivMon sketch and then describe these two approaches in more detail. In our implementation, we use the first approach to incorporate UnivMon.

- *Background on UnivMon sketch.* UnivMon is an ensemble of Count Sketches and consists of multiple levels. Each level maintains a Count Sketch. On every packet, a hash function is computed to decide a level and the corresponding level is updated.[3] The level is decided by the count of leading non-zero bits of the hash output. Each bit of the hash output is equally likely to be zero or one. Due to this, subsequent levels are accessed with exponentially decreasing probability, i.e.,

$$P(i) = \begin{cases} 2^{-i} & \text{if } i < k \\ 2^{-(k-1)} & \text{if } i = k \end{cases} \quad (2)$$

  where P is the probability of accessing level i, and k is the total number of levels.

- *Effective memory size accessed uniformly.* As shown in Figure 15, we study the variation of time per packet for the UnivMon sketch on a CPU as we vary its levels. We observe that UnivMon effectively behaves as if it had at most four levels, all of which are accessed uniformly. In the Optimizer, this corresponds to using T(Effective uniformly accessed memory) instead of T(Total memory) (T was defined in §4). For other sketches with complex control paths, a similar strategy can be used.

[3] Note, we use an optimized version of UnivMon described in [76] which is slightly different from the original UnivMon paper [11]

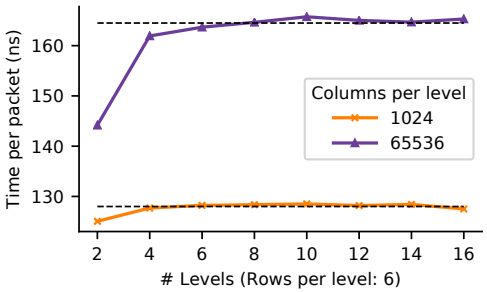


Figure 15: Dealing with non-uniform memory access patterns UnivMon effectively behaves as having at most 4 levels from the perspective of memory size accessed uniformly. The black dotted lines show the time per packet for a hypothetical sketch with $2*6+1=13$ hashes per packet and 6 memory accesses where the accesses are made uniformly. (The 13 hashes are: 2 hashes per row for a Count Sketch and one hash to decide level.)

- *Estimation of hit rates.* This is a more theoretical approach, but has similar results as above method. We assume that the caching mechanism on the device being profiled honors temporal locality to decide which items to keep cached, i.e., we can assume that the probability that an item is kept in the cache is proportional to its access frequency. Using this, we estimate the likelihood that an item is cached. Then using (1) the likelihood that an item is accessed (access frequency) and (2) the likelihood that the item is in cache; we estimate the expected inverse throughput of memory accesses leveraging inverse throughputs to different cache levels. Inverse throughputs to cache levels are estimated using the ridges in the memory benchmark of the Profiler (Figure 4a). In what follows, we illustrate application of this process on UnivMon with a toy device.

  Let's assume the UnivMon sketch has k levels and each level has consumes x bytes. For simplicity, let's assume the toy device has two cache levels: L1 and L2, with sizes x and $(k-1)x$ bytes respectively. Recall that, the ith level of UnivMon is accessed with probability $P(i)$. Through the locality principle, we expect that $P(i)$ fraction of level i would be present in L1 cache and $1-P(i)$ in L2 cache. Then the probability that an access goes to

L1 cache is:

$$
\begin{aligned}
P_{L1} &= \sum_{i=1}^{k} P_1 * P_2 \qquad (3)\\
&= P(i) * P(i)\\
&= \left(\sum_{i=1}^{k-1} 2^{-2i}\right) + 2^{-2(k-1)}\\
&= \frac{1}{4}\left(1 + \left(\sum_{i=2}^{k-1} 2^{-2(i-1)}\right) + 2^{-2(k-2)}\right)\\
&\approx \frac{1}{4}\\
\text{and, } P_{L2} &= 1 - P_{L1} \approx \frac{3}{4}
\end{aligned}
$$

where P_1 is the probability that level i of the sketch is accessed, P_2 is the probability that the accessed data is in L1 cache given that some data in level i is accessed, and P_{L2} is the probability that L2 is accessed. Then, the expected inverse throughput of memory accesses would be $P_{L1} * IT(L1) + P_{L2} * IT(L2)$, where $IT(.)$ is the inverse throughput to the corresponding cache level.

For the same toy device, we do a similar analysis for a sketch with size $4x$ bytes, where the memory accesses are made uniformly. Then random x bytes of the sketch would be in L1 and remaining $3x$ in L2. P_{L1} = probability that an element in L1 cache is accessed = $x/4x$. Hence, expected inverse throughput for memory accesses is $0.25 * IT(L1) + 0.75 * IT(L2)$. We see that the access probabilities (hit rates) and the expected inverse memory access throughput for this sketch is roughly equal for UnivMon example above, consistent with the empirical approach.

Such expressions for expected memory access time can be accommodated as constraints in the MIP formulation of the Optimizer albeit with increased solving time due to additional non-linear constraints. We leave exploration of this for future work.

Non-uniform Traffic. In the Optimizer and Profiler, we study performance and allocate resources for worst case (uniform) traffic. This is because the traffic distributions may not be known apriori, or one might want to allocate resources to handle adversarial cases. However, if this is not true, one could adapt our work to use the known traffic distribution to estimate hit rates to different cache levels similar to that described for accommodating sketches with non-uniform memory access patterns above.

Impact of profiling errors on Optimizer. We observe that most of the errors in profiling occur when sketches use a large amount of DRAM (or off-chip memory) on the devices. We seek to study what difference, such errors can create to the Optimizer's output. In the Optimizer's output, we identify devices which use a non-zero amount of DRAM, and whose resource allocation would change if the device profiles are off by 10%. We show in Figure 16 how many more resources would be needed for a Clos topology with 16 pods as we vary the sketch load (Y) (defined in §7.2). We observe that consistently less than 5% of devices satisfy the above conditions. Hence, the Optimizer's allocation would be off only for these 5% of devices. To illustrate this, let's assume all 5% devices are CPUs, each of these devices would need one more core if the profiles under-predict time per packet. Our savings suggest 0.5(50%) to 1(100%) cores saved per server. The errors are significantly smaller compared to the demonstrated savings. Note, that the profiles still are assumed to be accurate for the cases when sketches don't occupy DRAM.

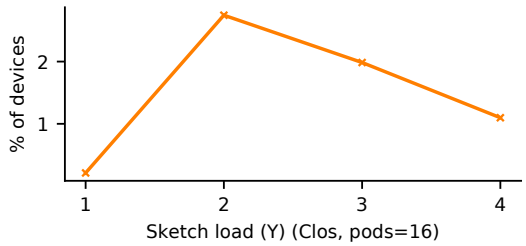


Figure 16: Impact of profiling errors The Y-axis shows the percentage of devices which use DRAM and whose resource allocation would change if the profiles are off by 10%.

Supplementary Evaluation. In addition to the Profiler evaluation for Count-Min sketches on CPUs, shown in Figure 6, we have also have the the detailed results for the other sketches including Count Sketch and UnivMon on CPU, SmartNIC, and FPGA shown in Figures 17, 18, and 19.

C Optimizer Details

Device profiles. Here we give an example of what the device profile ($dtime(.)$) looks like. The following shows the device profile for SoC SmartNIC.

$$
\begin{aligned}
dtime &= max\left(\frac{k_1 + u_h \cdot h}{c}, k_2 + u_m \cdot T(m)\right) \qquad (4)\\
&= max\left(dtime_h, k_2 + u_m \cdot T(m)\right)\\
&= max\left(dtime_h, k_2 + u_m \cdot t_{mem}\right)\\
dtime_h \cdot c &= k_1 + u_h \cdot h \qquad (5)\\
t_{mem} &= T(m) \qquad (6)
\end{aligned}
$$

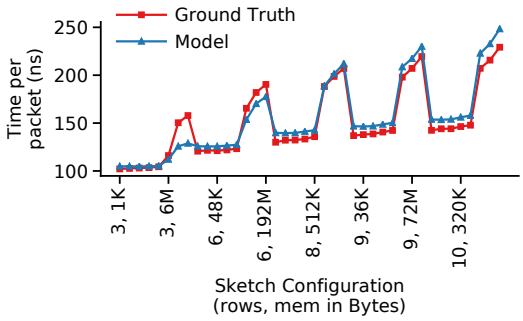
where u_h, u_m, and m (as defined in Section 4) are expressed as linear functions of sketch placement decision variables and c is a decision variable corresponding to the number of micro-engines. $T(.)$ is a non-linear function modelled using piecewise-linear constraints and the product terms: $u_m \cdot t_{mem}$, $dtime_h \cdot c$ are modelled using bi-linear constraints. These functions show a decoupled system with sketching done on the forwarding critical path with fraction of parallelizable execution $f \approx 1$.

Tweaking MI-BLP. We show how the MI-BLP formulation can be adapted to support other objectives/goals. We show in Equation 7, how a user can minimize performance overhead as an objective and subject to this minimum, again minimize resource overhead. This is done using hierarchical objectives [77].

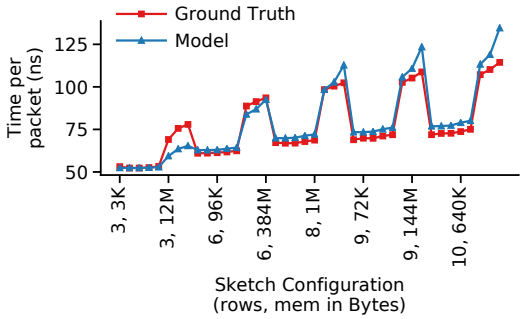
$$
\begin{aligned}
&\text{O1': perf} && \text{Minimize} \max_{d\in\mathcal{D}}(time_d), && (7)\\
&\text{O2': resources} && \text{Minimize} \sum_{d\in\mathcal{D}}(res_d + mem_d), \quad \textbf{s.t.}\\
&\text{C1: coverage} && \sum_{d\in p_\pi} b_{(d,s)} \geq 1 \quad \forall p\in\mathcal{P}, \forall s\in p_s\\
&\text{C2: accuracy} && mem_{(d,s)} \geq s_{mem}\cdot b_{(d,s)} \quad \forall s\in\mathcal{S}, \forall d\in\mathcal{D}\\
&\text{C3: capacity} && \sum_{s\in\mathcal{S}} b_{(d,s)}\cdot s_{rows} \leq d_{rows}, \quad \text{and}\\
& && mem_d = \sum_{s\in\mathcal{S}} mem_{(d,s)} \leq d_{mem} \quad \forall d\in\mathcal{D}\\
&\text{C4: profiles} && \forall d\in\mathcal{D}:\\
& && time_d = d_{time}(res_d, \mathcal{P}_d,\\
& && \qquad \{(mem_{(d,s)}, b_{(d,s)}) | s\in\mathcal{S}\}\\
&\text{C5: traffic} && time_d \leq \frac{1}{d_{\text{traffic}}} \quad \forall d\in\mathcal{D}, \quad \text{where}\\
& && d_{\text{traffic}} = \sum_{p\in\mathcal{P}_d} p_t, \quad \mathcal{P}_d = \{p | d\in p_\pi, p\in\mathcal{P}\}
\end{aligned}
$$

Supplementary Evaluation.

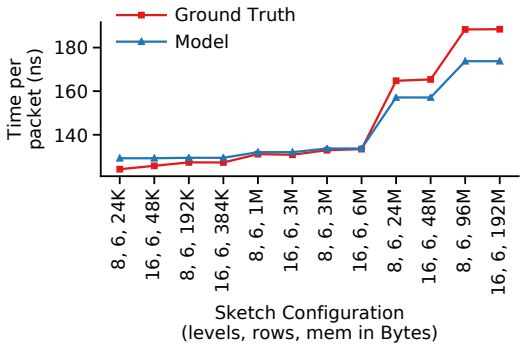
- We show additional metrics collected for Figures 7a and 7b in Figures 20& 21.
- One of the ways in which HeteroSketch reduces resource overhead is through efficient use of resources that it allocates. We see in Figures 8a, 8a, and 8a, that HeteroSketch overall allocates less number of resources but better utilizes each resource that it does allocate including CPU cores, SoC NIC memory bandwidth, micro-engines on the SoC smart-NIC.



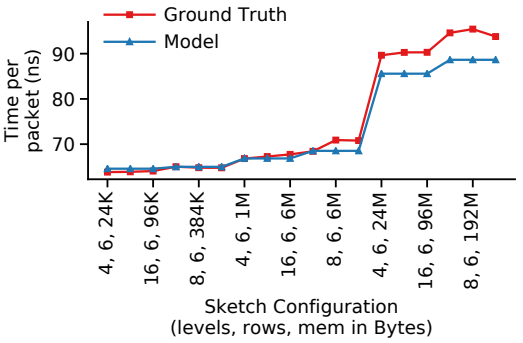
(a) 2 Cores, Count Sketch



(b) 4 Cores, Count Sketch

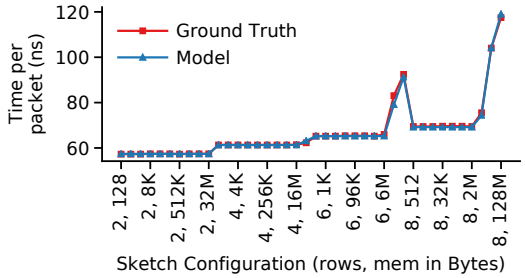


(c) 2 Cores, UnivMon

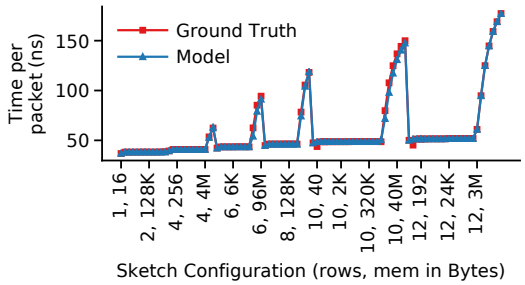


(d) 4 Cores, UnivMon

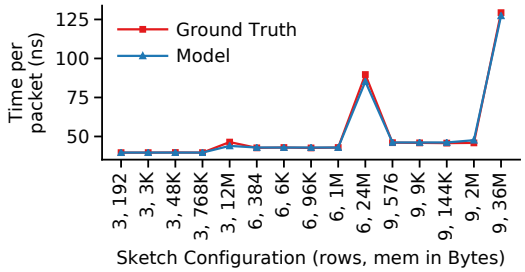
Figure 17: Performance Profiler — CPU Model



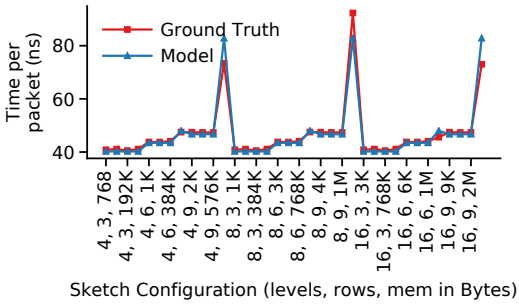
(a) 36 Micro-engines, Count-Min Sketch



(b) 54 Micro-engines, Count-Min Sketch

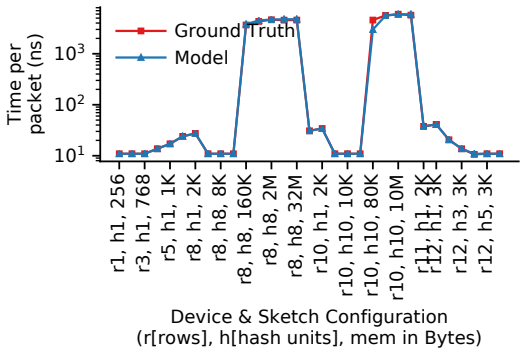


(c) 54 Micro-engines, Count Sketch

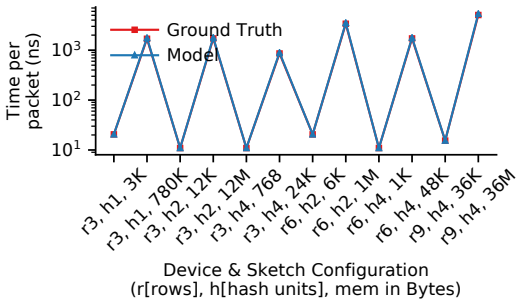


(d) 54 Micro-engines, UnivMon

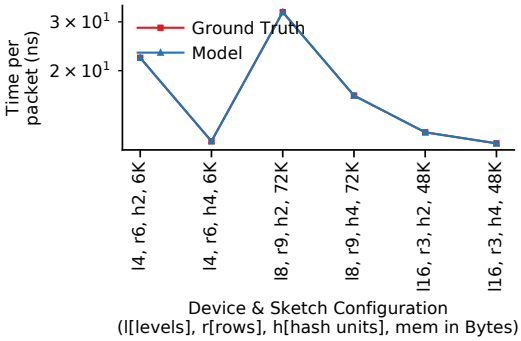
Figure 18: Performance Profiler — SmartNIC Model



(a) Count-Min Sketch

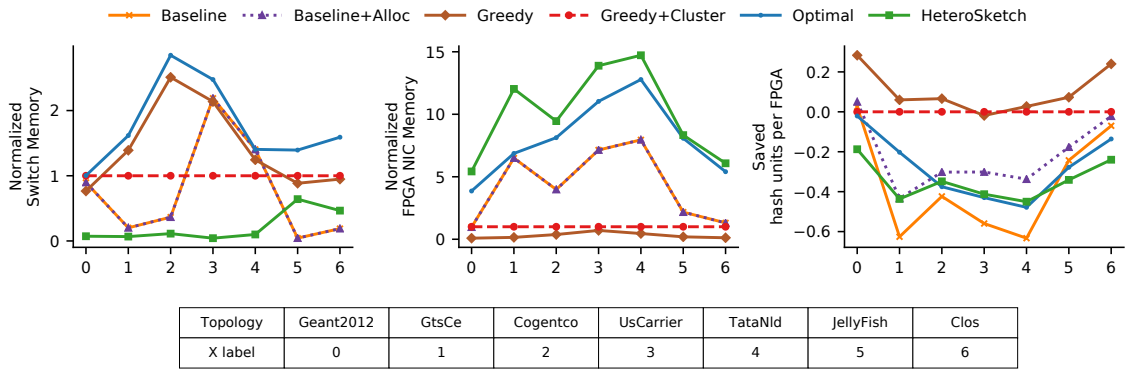


(b) Count Sketch



(c) UnivMon

Figure 19: Performance Profiler — FPGA Model (Note: Y-axes are in log-scale)



Topology	Geant2012	GtsCe	Cogentco	UsCarrier	TataNld	JellyFish	Clos
X label	0	1	2	3	4	5	6

Figure 20: Supplementary Optimizer Evaluation — [Topology Study] These figures show the difference in resource usage for experiments in Figure 7a in terms of switch & FPGA memory, and FPGA hash unit instances. Compute resources are shown in terms of amount saved relative to Greedy+Cluster (negative compute resources implies more resource consumption than Greedy+Cluster). Total resources and memory resources are normalized w.r.t Greedy+Cluster.

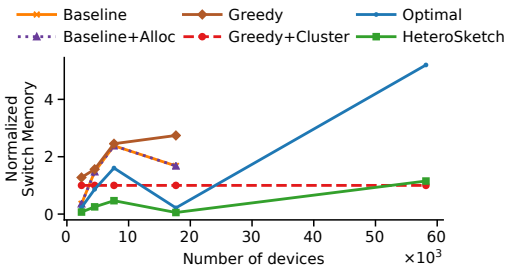


Figure 21: Supplementary Optimizer Evaluation — [Scale Study] These figures show the difference in resource usage for experiments in Figure 7b in switch memory normalized by that of Greedy+Cluster.