BALANCING PERFORMANCE AND FLEXIBILITY IN HYBRID NETWORK TELEMETRY SYSTEMS

John Sonchack

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

Jonathan M. Smith
Supervisors of Dissertation

Rajeev Alur
Graduate Group Chairperson

Dissertation Committee

Boon Thau Loo

André DeHon

Vincent Liu

Jennifer Rexford

ABSTRACT

BALANCING PERFORMANCE AND FLEXIBILITY
IN HYBRID NETWORK TELEMETRY SYSTEMS

John Sonchack

Supervisors: Jonathan M. Smith

Network statistics collection, calculation, and reporting, referred to as network telemetry, faces the problem of competing performance and flexibility demands. This thesis proposes a novel approach to balancing these requirements. Tensions between design goals are overcome by exploiting hardware heterogeneity, where hardware elements fill specialized roles and careful software structuring maps activities to appropriate hardware.

The axes of performance addressed here are coverage, throughput, and cost. Coverage can be gauged by the number of concurrent flows a telemetry system can measure. High coverage is important for application correctness, but due to the fundamental trade off between memory latency and size it is often sacrificed in favor of higher throughput or lower cost. I show that combining specialized and general purpose processors in a way that exploits locality can yield a somewhat surprising result, namely that high coverage and throughput can be achieved at low cost.

Flexibility describes a telemetry system's capability to support diverse application needs, including concurrent measurement and late binding of potentially complex metrics. A small reduction in performance greatly increases flexibility. By enhancing system components to summarize data in an efficient universal format, a hybrid telemetry system can lift metric calculation to software for flexibility by design, while still exploiting workload properties and hardware heterogeneity for high performance.

Validation and characterization of underlying principles is the third contribution. The major system elements have been validated with terabit rate implementations and simple but accurate performance models generalize results and guide future work.

Contents

Chapter 1

Introduction

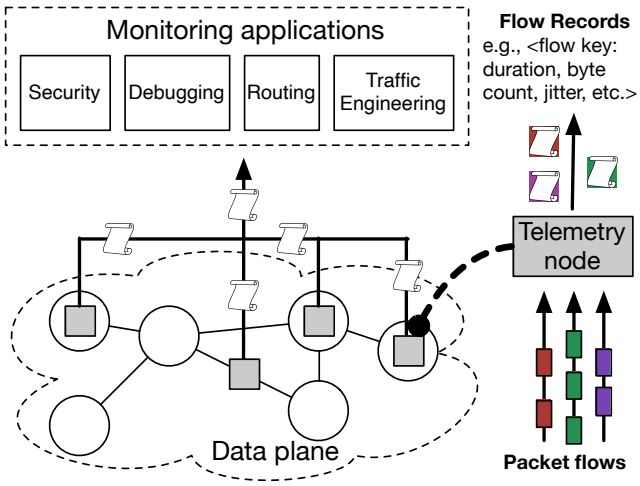

Figure 1.1: Telemetry systems summarize data plane traffic into flow records.

	Flow 1	Flow 2
Flow Key		
Source IP	10.1.1.1	10.1.1.6
Dest. IP	10.1.1.2	10.1.1.7
Source Port	34562	12520
Dest. Port	80	88
Protocol	`TCP`	`UDP`
Flow Metrics		
Duration	5s	100ms
Total length	88647B	3452B
Average jitter	1ms	0.1ms
Mean pkt. length	1401	342

Figure 1.2: Example TCP flow records with 4 simple metrics.

Telemetry is the capability for a network to measure and report information about the data that it carries. As Figure 1.1 illustrates, this capability is implemented as a telemetry system distributed across a network's data plane. Nodes in a telemetry system, which can be agents on a router/switch or standalone middleboxes, summarize packet flows (e.g., all packets belonging to one direction of a `TCP` connection) into *flow records* (FRs). Table 1.2 shows an example of flow records.

Telemetry systems are widely deployed because visibility into the network can help solve many problems. A recent survey of network operators, for example, reported that over 70% of the participants had telemetry capable data planes [1]. The earliest proposed application of network telemetry was to support management tasks such as provisioning, auditing, and billing [2]. Seminal work [3] identified many others. Since then, research has grown exponentially [4] and demonstrated a range of applications including security [5], traffic engineering [6, 7], profiling [8], and visualization [9, 10].

These new and increasingly diverse applications create demand for more flexibility in telemetry systems. For example, recent applications depend on flow metrics that are domain specific [11] and potentially complex [12]. It has proven challenging to balance such flexibility requirements with performance, which has always been critical for telemetry systems due to the high volume of traffic that they must measure.

Telemetry systems designed for flexibility target general-purpose architecture [13, 14], e.g., servers, with large memories and the capability to calculate arbitrary metrics. However, inherent bottlenecks such as I/O bandwidth and memory latency limit performance.

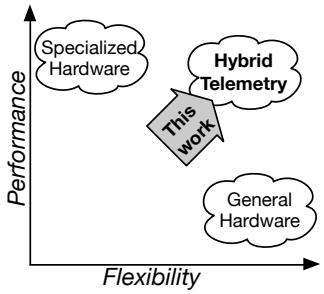

Figure 1.3: Solution space.

Performance-oriented telemetry systems [11, 15, 16] target architectures specialized for high throughput. For example, *line-rate* programmable forwarding engines (PFEs) that process one packet per cycle [17, 18, 19]. Unfortunately, specializing for high and guaranteed throughput restricts computation [17], which limits not only flexibility but also axes of performance besides throughput and cost.

As Figure 1.3 illustrates, this dissertation introduces a hybrid solution that balances performance and flexibility by combining line-rate and general-purpose architectures. The key is carefully dividing system activities to get the best of each platform. The thesis is: *coupling a line-rate front-end processor for customizable data extraction and gathering with a general-purpose back-end processor for arbitrary metric calculation can exploit workload properties to balance performance and flexibility.*

Design goal	Motivation
Performance	
Coverage	Application correctness.
Throughput	High link speeds.
Cost	Network density and scale.
Flexibility	
Concurrency	Many potential applications.
Late metric binding	Dynamic applications.
Unrestricted metric programmability	Problem specific and complex metrics.

Table 1.1: Telemetry system design goals.

1.1 Design Goals

Telemetry systems face a challenging set of performance and flexibility related design goals that arise from their operating environment and the needs of the applications that they support. Table 1.1 summarizes these design goals, which this section describes in more detail.

Coverage. Coverage quantifies the percentage of the underlying traffic accounted for by the telemetry system. Coverage is relevant at the flow level, i.e., what fraction of flows has the telemetry system measured, and at the metric level, i.e., what fraction of packets in a flow has the telemetry system measured? Coverage is important for application correctness and accuracy [20, 21].

Throughput. Throughput is the maximum rate of traffic that a telemetry system can measure, gauged in Gb/s. It is important because link and switch bandwidths are high and ever-increasing. For example, a top-of-rack switch in 2019 forwards at multiple terabits per second. These high potential bandwidths, combined with the demand for high coverage, necessitates that a telemetry system support high throughput.

Metric	Purpose
Concurrent connection count	Is there a denial of service attack?
EWMA latency	Is there a micro-burst?
Flowlet size histogram	Is my load balancer working well?
TCP out of sequence count	Is the network reordering packets?
Packet counts per source	Who is using the network most?

Table 1.2: Example flow metrics, from [11], and the questions that they answer.

Cost. The cost of a telemetry system is gauged in resources per B/s of traffic. Resources can be network resources such as switch memory, bandwidth, and processing servers; or economic resources such as dollars and Joules. Low cost is important for a telemetry system to be practical in large scale networks, e.g., with thousands of switches.

Concurrency. Concurrency is a telemetry system's capability to support the measurement needs of multiple monitoring applications at once. Concurrency is important because often, solving a network problem depends on engineering the right metric. For example, consider the five simple metrics listed in Table 1.2. Each metric helps a user or system answer a different and important question about the network. In a large or shared network, independent entities may want to measure all of these metrics, and many more, at the same time.

Late metric binding. The binding time of a metric is the point at which its calculation function is linked to the telemetry system. For many scenarios, it is beneficial to bind metrics after the telemetry system has compiled, either at or after runtime.

- **Runtime binding** means being able to update or change metrics while the telemetry system is operational. It allows metrics to be changed much more rapidly, which enables powerful dynamic applications. For example, interac-

tive diagnostics platforms [11], where users define custom metrics specific to the problems they wish to debug. Runtime metric binding also compliments concurrency by making it easier to change the set of active applications.

- **Post-runtime binding** means being able to change metrics after telemetry data has been collected. This can simplify the development of new metrics and applications. For example, some flow metrics are calculated by applying machine-learning models [12]. Although the models themselves ultimately run online (e.g., attributing a label to each flow as), they are often easier to engineer, optimize, and train offline, from previously collected data.

Unrestricted Metric Programmability. Telemetry systems best serve applications when they can be programmed to calculate arbitrary metrics, i.e., any metric that can be evaluated on a general-purpose platform like a server. This is because metrics are often engineered for specific problems and can be surprisingly complex. Consider the example systems below, which rely on metrics that can potentially require large amounts of state, many instructions, advanced numerical analysis, and multiple iterations over data to compute.

- Intrusion detection systems calculate *maliciousness scores* for flows by applying sets of complex rules [22] or machine learning (e.g., neural networks [23]).

- Traffic classifiers compute *application labels* that identify which application is most likely to have generated a flow. These labels derive from complex numerical functions, for example, a Fourier transform of packet inter-arrival times [12].

- Diagnostics tools help identify the root cause of a fleeting network event, e.g., high queue lengths 3.6, by assigning significance scores to flows. These scores quantify the correlation between the flow and the event, and are computed by replaying the sequence of packet arrivals immediately preceding the event.

Processor	**CPU (Server)**	**PFE (Switch)**
Design		
Performance goal	best effort	guaranteed high throughput
Intended task	general-purpose	real-time packet processing
Performance		
I/O bandwidth	100 Gb/s	> 1000 Gb/s
Memory latency	1-100 ns	1 ns
Flexibility		
Memory access	general	restricted
Memory size	>1000 MB	16 MB
Program complexity	unbounded	bounded

Table 1.3: Properties of CPUs and PFEs that are relevant to performance and flexibility.

1.2 Homogeneous Telemetry Systems

A homogeneous telemetry system is one that concentrates core system tasks, such as grouping packets into flows and calculating metrics, into a single type of processor. Today, most telemetry systems are homogeneous, designed for either general-purpose processors [13, 14], e.g., commodity servers with CPUs, or processors specialized for guaranteed high throughput [15, 11, 24], e.g., switches with fixed-function measurement ASICs [25] or programmable forwarding engines (PFEs) [18, 17].

Homogeneity makes it challenging to balance performance and flexibility because general processors limit performance while specialized processors limits flexibility. Table 1.3 lists a few important properties of CPUs and PFEs, the remainder of this section explains how they impact performance and flexibility.

Limitations of CPUs (Servers)

For telemetry, servers have limited performance because of two fundamental bottlenecks: high memory latency and limited I/O throughput. High memory latency is a

result of a large memory. Cache hierarchies can reduce average latency, but in many cases high average latency is unavoidable, e.g., tasks that access a large amount of state with random patterns.

I/O bottlenecks are a result of the complex subsystems that lie between the network and a server's main memory. Network cards and system buses are orders of magnitude slower than a network switch [26], around 50 Gb/s per network card versus around 1 Tb/s per switch.

These bottlenecks limit the per-node throughput of server-based telemetry. For example, commercial telemetry appliances [27, 28] support around 40 - 100 Gb/s, compared to over 1 Tb/s for a top-of-rack switch.

As a result of the limited per-node throughput, general-purpose hardware makes users choose between cost and coverage. Higher throughput can be achieved by scaling up to many servers, but this increases cost. Alternately, it is always possible to achieve a higher effective throughput by only monitoring a fraction of flows [29] or a fraction of the packets in each flow [14, 30, 29], but this sacrifices coverage.

Limitations of PFEs

PFEs limit a telemetry system's flexibility because of restrictions on memory and computation. These restrictions are unavoidable, arising from the power, chip space, and timing budgets [17] that must be met to guarantee high line rates at low cost.

Memory. Memories in a PFE are small and their access is restricted to enable high predictable throughput. For example, the highest throughput PFEs, reconfigurable ASICs, process 1 packet per cycle with a 1 Ghz clock rate [18, 17] and have around 16 MB of SRAM. The SRAM is divided into program defined memories, each of which can only be accessed once per packet to fit the 1 cycle budget. The small capacity makes it challenging to support high coverage and concurrent measurement. In addition, the restricted access model limits the space of metrics that can be computed.

Computation. Since their time budget is fixed, PFEs can only apply a limited number of operations to each packet. Reconfigurable ASICs are the most restrictive, especially with respect to sequential code. They process packets with a strict feed-forward pipeline of limited depth, e.g., a pipeline of around 16 stages [18, 17]. It is not possible to move backwards in the pipeline, so the maximum sequential code length is equal to the pipeline depth. These restrictions bound the space of metrics that a PFE can evaluate.

In a PFE, program instructions are stored in small on-chip memories so that they can be accessed quickly. The small instruction store limits a telemetry system's capability to support concurrent metrics. Further, changing metrics at runtime is challenge because updating the instruction store is disruptive. As Chapter 3 discusses, reloading a PFE can pause all switch functions (including forwarding) for multiple seconds. Even in the best case, where the PFE hot-swaps to a second physical pipeline or instruction store with updated code, there are challenging open questions, e.g., maintaining consistent state.

1.3 This Dissertation: Hybrid Telemetry Systems

This dissertation introduces a hybrid approach to telemetry, where the system is carefully split between a CPU and PFE to balance performance and flexibility. Clearly the characteristics of the processors are complementary, with PFEs supporting high throughput and CPUs providing general compute capabilities. It is less clear, however, whether they can be combined to achieve high performance without sacrificing flexibility.

The thesis of this dissertation is: ***coupling a line-rate front-end processor for customizable data extraction and gathering with a general-purpose back-end processor for arbitrary metric calculation can exploit workload properties to balance performance and flexibility.***

The following chapters present systems and models that validate this thesis.

Workload Properties

There are two common workload properties that guide the design of a high performance and flexible hybrid telemetry system: high locality and small feature sets.

High locality. Locality is the tendency of a system to reference the same data locations repetitively over a short period of time. In a telemetry system, references are to flow records and the characteristics of a multi-flow packet stream result in a high degree of locality. For example, endpoints tend to transmit packets in bursts [31], so when a telemetry system observes a packet from a flow it is likely to observe more packets from that flow again in the near future.

A high degree of locality motivates the use of caches – fast but small memories that amortize the cost of accessing a larger but slower main memory. The cache stores recent items. Slower main memory is only accessed for items not in the cache. A hybrid telemetry system can leverage caches in two ways: first, it can leverage the PFE's memory as a custom cache for relevant state. Second, its data structures can be optimized to take advantage of a CPU's cache hierarchy.

Small Feature Sets. Each group of flow metrics derives from a set of packet features that is small, relative to the size of the total packet. Additionally, the feature set is often compact with respect to the flow metrics it generates, i.e., a small number of features generate a large number of metrics. For example, total flow duration, average packet interarrival time, and average jitter all derive from a generating set with 1 field: packet arrival timestamp.

The small size of feature sets motivates early filtering of un-necessary header fields to alleviate I/O bottlenecks. The compactness of feature sets implies that the cost of storing the per-packet data necessary to generate a set of metrics is low. This gives the telemetry system freedom to delay metric calculation, e.g., to a more flexible back-end platform, without adding unreasonable overhead.

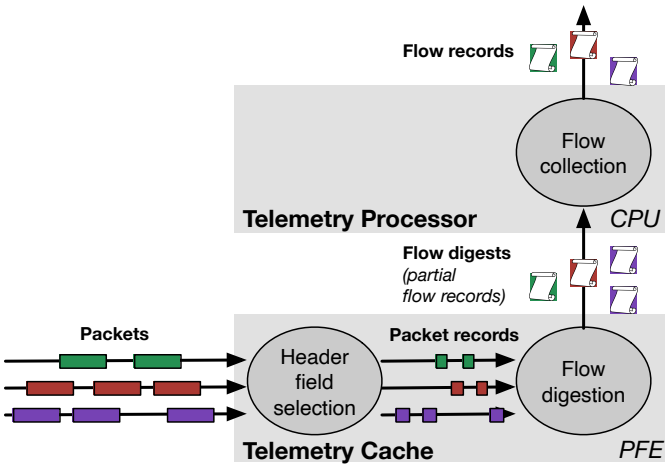

Figure 1.4: High level architecture of TurboFlow.

Contributions

This dissertation introduces working terabit-rate hybrid telemetry systems and models that show how to exploit the above workload properties to balance performance and flexibility goals.

TurboFlow. The first system, TurboFlow, balances performance and coverage. This requires low average memory access times, to keep throughput high at low cost, and also a large memory, to store records of many concurrent flows. TurboFlow meets these competing requirements by structuring the system as a *telemetry cache* in the PFE and a *telemetry processor* in the CPU, as Figure 1.4 illustrates. The telemetry cache is tasked with *field extraction* and *flow digestion*, while the telemetry processor handles *metric storage*.

- First, the PFE-based telemetry cache extracts the flow key and relevant features from each packet's header and places them into a packet record. All downstream components operate on these small records instead of the underlying packets. This effectively eliminates I/O bandwidth as a bottleneck because packet features are small compared to packets.

- Second, the telemetry cache uses a small amount of PFE memory to transform

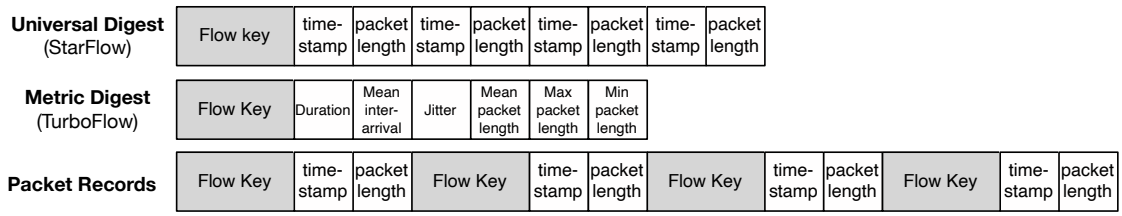

Figure 1.5: Comparison of digests exported by TurboFlow and StarFlow.

each flow's packet records into a stream of flow digests. A digest summarizes one flow over the span of several (e.g., 2 - 32) of its most recent packets. Like packet records, digests need to be aggregated into complete flow records. However, processing a digest stream is less work because the cost of expensive operations (e.g., memory operations) is amortized across all the packets that each digest summarizes.

- Finally, the CPU-based telemetry processor aggregates flow digests into complete flow records. The cost of main memory operations are further amortized by data structure optimizations that maximize hit rates of the telemetry processor's CPU caches. The combination of these data structure optimizations and the PFE-based telemetry cache increases the effective throughput of the telemetry processor by orders of magnitude.

By eliminating I/O overheads and amortizing the cost of memory operations, TurboFlow allows a single PFE and low power CPU to achieve full coverage of terabit-rate traffic with tens of millions of concurrent flows at significantly lower cost than prior systems (Table 2.13).

StarFlow. Second, this thesis introduces StarFlow, a telemetry system that adds support for concurrent measurement, late binding, and complex metrics in exchange for a small reduction in performance. The high level structure is similar to TurboFlow: the PFE serves as a telemetry cache while the CPU serves as a back-end processor.

The change that lets us make a principled trade between performance and flexibility is the format of digests exported from the telemetry cache to the telemetry processor. Figure 1.5 illustrates the difference. Instead of a set of metrics, a StarFlow digest contains a batch of packet records from a single flow. Each packet record contains the header fields required to derive the flow metrics. The digests are *universal*, in the sense that they can also be used to calculate any other metrics that derive from the same fields.

Changing the interface between the telemetry cache and processor allows the processor to leverage its general compute capabilities to support flexibility goals. It can support concurrent measurement, by either time-multiplexing a single CPU core or scaling out to per-application cores, change metrics at runtime without disruption, and be programmed to calculate complex metrics that a PFE could not support.

This added flexibility comes at a performance cost because universal digests reduce the amount of amortization provided by the PFE and increase the CPU's workload by tasking it with metric calculation. However, StarFlow's overall performance is still high because: 1) the universal digests are compact enough to eliminate the CPU's I/O bottleneck and 2) the CPU's main bottleneck is memory operations related to flow lookups, which the universal digests still amortize.

Benchmarks in Chapter 3.7 show that StarFlow allows a single PFE and server-class CPU to achieve full coverage of terabit-rate traffic while supporting many concurrent metrics, late binding, and metrics that require general compute capabilities such as large memories, multiple processing iterations over large buffers, and advanced numerical operations.

Analytic Performance Model. The final contribution is a framework for understanding hybrid telemetry system performance, based on simple but accurate analytic and stochastic models. These models characterize the relationship between network traffic workloads, measurement tasks, and hybrid telemetry system performance. By applying the models, this thesis generalizes the evaluations of `TurboFlow` and `*Flow`,

quantifies the effects of key design choices in these systems, and identifies ways that future work can further improve performance.

1.4 Outline

The remainder of this thesis describes the above contributions.

- Chapter 2 introduces `TurboFlow`, a hybrid telemetry system that exploits locality and the small size of feature sets to achieve high performance and flow coverage.
- Chapter 3 extends the design of `TurboFlow` with *universal digests*, a flexible intermediate format for telemetry data. Based on the idea it introduces `*Flow`, a hybrid telemetry system that supports concurrency, late binding, and advanced metrics.
- Chapter 4 develops the model-based framework and uses it to generalize results and compare `TurboFlow` versus `*Flow` in Internet and datacenter scenarios.
- Chapter 5 surveys related work.
- Chapter 6 concludes, summarizing the contributions of this thesis.

Chapter 2

Achieving Throughput and Coverage

2.1 Introduction

High coverage, the capability to measure a large fraction of the packets and flows in a network, is important for the correctness and accuracy of many applications [20, 21].

- Security systems monitor every flow in a network, to detect stealthy or low volume attacks [32].
- Debuggers measure flows from multiple points in the network simultaneously, to localize network faults [15].
- Load balancers analyze the demand of all flows using shared paths, to adapt routes and reduce congestion [6].
- Researchers profile networks to test new ideas and systems [33, 34, 35].

Rather than measure traffic and switch state directly, which would not scale, applications that need high coverage rely on telemetry systems to stream flow records (FRs) up from the data plane [11, 15, 36]. A FR summarizes a fine grained `TCP/UDP`

flow with metrics describing the packets in the flow and the network conditions they observed. FRs give applications visibility into the network, but are also compact enough to collect and analyze at large scales. They are compact because each FR summarizes all the packets in a flow with fixed width aggregate metrics. For example, FR traces are 2-3 orders of magnitude smaller than corresponding packet [37] traces.

Although FRs are a powerful abstraction for applications, FR telemetry systems that support high coverage are prohibitively expensive. For example, server-based appliances are rated for around 100 Gb/s of traffic per node [27, 28], bottlenecked by I/O limitations and high memory latency. In large networks with thousands of switches [33, 34, 35], each capable of forwarding traffic at multi-terabit rates, a server based telemetry infrastructure would comprise many racks of servers.

To reduce costs, telemetry systems employ network switches with line rate forwarding engines to either sample packets [14] (allowing each back-end measurement server to cover more flows) or measure flows directly [11] (eliminating the need for back-end measurement servers). Both strategies limit coverage.

- Sampling limits packet coverage because each telemetry appliance only has enough throughput to measure a fraction of the packets a switch forwards.
- Forwarding engine measurement limits flow coverage. Forwarding memories use small memories to achieve guaranteed high throughput, so they only have enough capacity to track a small number of concurrently active flows.

Today, the only FR telemetry systems that can achieve high coverage to ensure application correctness and accuracy (Section 2.2), are server-based and prohibitively expensive to deploy at scale.

Introducing TurboFlow. Motivated by the desire for high coverage FR telemetry at low cost, this chapter introduces `TurboFlow`. `TurboFlow` is a FR generator for commodity programmable switches that can measure every packet and flow in terabit-rate traffic using only a switch's PFE and management CPU. It can be deployed on

commercially available switches [38], in parallel with other forwarding and routing processes. `TurboFlow` also supports limited metric programmability, with a class of application-defined metrics similar to other recent telemetry systems [11].

`TurboFlow`'s combination of high throughput, coverage, and programmability make it a low cost and powerful solution that enables dozens of monitoring applications [39, 40, 5, 41, 42, 43] at unprecedented scales.

Design. Achieving high throughput without sacrificing coverage is challenging because in general, the table containing per-flow measurements must be stored in the switch's main memory since it is too large to fit anywhere else. The same bottlenecks that limit server-based telemetry systems arise: limited I/O bandwidth to memory and high access times. `TurboFlow` overcomes these bottlenecks by exploiting the heterogeneous hardware elements in a programmable switch and the general properties of a telemetry workload.

First, to overcome the limited I/O bandwidth, `TurboFlow` uses the PFE's programmable line rate parsers and VLIW units [18] for *selection*: extracting the packet data relevant to telemetry and packing it into a compact digest format that contains partially calculated flow metrics. This significantly reduces the amount of data that needs to move from the switch's PFE to its CPU and main memory because the digests are much smaller than the packets that they summarize. For example, a 64B digest can summarize multiple 1500B packets.

Second, `TurboFlow` amortizes the high cost of memory operations by caching state in small but fast on-chip memories in the switch's PFE and CPU. The PFE component of `TurboFlow` uses programmable SRAM banks and stateful ALU primitives [17] to calculate and cache metrics for recently active flows. Instead of sending a record for every packet up to the CPU, the PFE sends digests that each summarize multiple recent packets. Typical packet counts range from 10-100 (Section 2.7) depending on workload. This amortizes the cost of all CPU processing, including memory operations. Additionally, the CPU component of `TurboFlow` uses data structures that

leverage on-chip memories (registers and caches) to further reduce the frequency of main memory operations. Caching is extremely effective for a telemetry system because of the inherent locality in network traffic, as Chapter 4 explains.

Evaluation. Two proof of concept implementations, targeting switches with significantly different PFE architectures, validate `TurboFlow`'s design. The first implementation targets the Wedge 100BF-32X [38], a 32x100 GbE switch with a Tofino [19] PFE; the second targets a 4x10 GbE prototype switch using a NFP-4000 PFE [44]. On both platforms, `TurboFlow` scales to monitor all flows on all links with workloads from Internet and simulated data center traces.

Benchmarks show that `TurboFlow`'s PFE component increases the effective throughput of the switch's CPU by a factor of 10-100, while its data structure optimizations further improve throughput by an additional factor of 20 (Figure 2.11. Based on analysis of the benchmark results, `TurboFlow` makes programmable high coverage telemetry cost effective in Internet and data center scenarios. Compared with other recent PFE accelerated telemetry systems, `TurboFlow` reduces the equipment and power cost of generating FRs by a factor of more than 5, as Section 2.7 shows.

Contributions. This chapter makes 4 contributions.

1. An analysis of the trade offs between throughput, coverage, and cost in telemetry systems.

2. `TurboFlow`, a FR generator that exploits hardware heterogeneity and general workload characteristics to support high coverage, high throughput, and limited programmability at low cost.

3. Proof of concept implementations of `TurboFlow` for two programmable switches.

4. A thorough evaluation and cost analysis of `TurboFlow`, demonstrating its benefit in large networks.

	Flow 1	Flow 2
Flow Key		
Source IP	10.1.1.1	10.1.1.6
Dest. IP	10.1.1.2	10.1.1.7
Source Port	34562	12520
Dest. Port	80	88
Protocol	`TCP`	`UDP`
Flow Metrics		
Packet Count	5	7
Byte Count	88647	3452
Max Queue Length	0	34
Avg. End-to-end Latency (us)	10	300

Table 2.1: Example flow records.

2.2 Background on Flow Record Telemetry

Flow records (FRs), depicted in Table 2.1, compactly summarize information about packet flows and how they were processed by the network. `TurboFlow` focuses on FRs that aggregate packets at the level of IP 5-tuple, i.e., by TCP connection or UDP stream. FRs are commonly referred to as *Netflow* [45] or IPFIX [46] records and used by many applications, as Table 2.2 shows.

FRs are an appealing record format because they are much smaller than the flows they summarize, which makes network-wide monitoring practical. For example, an hour-long packet trace from a 10 Gb/s Internet router link would contain nearly 1 TB of data and over 1 billion packets [47]. At such high rates, it is not practical to collect or analyze data from more than a hand full of links. On the other hand, a FR trace that summarizes important statistics about each flow using the record format depicted in Table 2.1 is around 5 GB with 50 million records. This represents a 20X reduction in processing rate and a 200X reduction in bit rate for the application. At these much lower rates, an analysis application implemented on an efficient general purpose stream processing platform [48] could monitor *hundreds* of 10 Gb/s links

Feature Type	Examples	Applications
Header-based Features		
Categorical Features	QoS type, IP options, TCP options & flags	Security [5], flow scheduling [6, 51], auditing [43], heavy hitter detection [24], QoS monitoring [11]
Metric Features	duration, packet count, byte count, jitter, max packet size	
Metadata-based Features		
Categorical Features	ingress port, egress port, selected route	Loop and black hole debugging [15], performance queries [11], load balancing [52], network design [33]
Metric Features	max queue depth, avg. latency, dropped packet count	

Table 2.2: Types of FR metrics and example applications.

with a single server.

FRs are also appealing because they summarize traffic at the level of individual TCP or UDP streams. The fine granularity preserves information that is important for many applications. For example, host communication patterns that reveal botnets [49] or other attacks [5] and per-connection packet counts that make it easier to localize misconfiguration in the network core [15] and engineer traffic [50].

As Table 2.2 shows, a FR can also contain categorical features about flows, in addition to metrics. Many features derive from packet header fields, to give applications visibility into the characteristics of network traffic. Other features derive from metadata that describes how a packet was processed, which is typically only available in switch-based telemetry systems. These metadata derived features give applications visibility into the operation of the network itself, rather than the traffic it carries.

Telemetry Switch Design Goals

Telemetry as a switch feature reduces the cost and complexity of network monitoring, compared to server-based telemetry. There are three important design goals for telemetry switches: programmability, throughput, and coverage.

Programmability. Programmability describes the capability for monitoring applications to define the metrics included in FRs. It is important because often, solving a problem depends on engineering the right metric. For example, consider bot detection [49], QoS optimization [50], and incast debugging [11]. Bot detection systems analyze communication graphs between hosts that derive from simple metrics like per-flow byte counts and timestamps. On the other hand, QoS optimization can require more exotic metrics that describe network performance, such as dropped packet counts and path delays. Incast debugging requires completely different metrics that describe internal operation of the switch, e.g., queue depth.

This chapter focuses on programmability with respect to a limited class of metrics: decomposable aggregation functions [53] that can be calculated in a line rate switch forwarding engine.

A decomposable aggregation function is one that can be calculated by splitting data into an arbitrary number of subsets, applying a calculation function to each subset, and then merging the resulting values. Any commutative and associative function is trivially decomposable. This includes the average, minimum, and maximum of any packet header field or metadata associated with processing, all of the metrics listed in Tables 2.1 and 2.2, and all statistics supported by current Netflow ASICs.

This is the same subset of metrics that other recent high performance telemetry systems target [11]. Chapter 3 describes how to support programmability of more general metrics.

Coverage. There are two axes of coverage: packet coverage and flow coverage.

Packet coverage is the fraction of packets accounted for in each measured flow. Low packet coverage reduce the accuracy and correctness of many applications. For example, consider traffic load balancers [6, 51] that re-route flows to maximize network bisection bandwidth. Low packet coverage reduces the accuracy of flow throughput estimation [24], which can cause a load balancer to identify the wrong flows for re-routing.

Flow coverage is the fraction of traffic flows measured with FRs. It is important because some applications require high coverage. For example, intrusion detection systems can miss attacks or underestimate the magnitude of a DDoS if they only see a sample of flows [54, 20, 30]. Additionally, high flow coverage better supports late binding. By summarizing *all* flows into compact FRs, a telemetry system gives monitoring applications the opportunity to choose what flows they monitor at runtime. This is especially important in a large network where the telemetry system supports many applications and potentially unforeseen use cases, as it may not know which flows are important a-priori.

Throughput. Throughput describes the rate of traffic that a telemetry switch can measure. High throughput is important because switch forwarding rates are high. For example, a top of rack switch in 2019 can forward over 1 Tb/s of data per second [38]. If the telemetry feature of a switch cannot keep up with these rates, it will need to ignore a subset of traffic, which reduces coverage, or rely on monitoring servers to cover a subset of links, which adds cost.

Given the low per-node throughput of monitoring servers, e.g., 100 Gb/s [27, 28], even relying on servers to cover a few links per switch can drastically increase the total power and equipment cost of a data plane. In practice, operators [55] and researchers [51, 56, 57] go to great lengths to minimize these costs.

	Throughput	High coverage	Programmable aggregation
Specialized ASIC			
Packet Sampling	✓	✗	✓
Netflow ASICs	✓	✗	✗
PFE Accelerated			
FlowRadar	✓	✗	✗
Marple	✓	✗	✓
TurboFlow	✓	✓	✓

Table 2.3: Comparison with prior switch FR generators.

Prior Telemetry Switches

FRs have a long history of use [43] and many telemetry switch solutions have been developed [37]. At a high level, systems can be classified into two broad categories based on the type of switch hardware that they employ. Older systems are based on fixed-function switch hardware, while newer systems use reconfigurable switch hardware, i.e., programmable forwarding engines (PFEs). Table 2.3 summarizes the solution space with respect to the design goals of throughput, coverage, and programmability.

Sampling. Sampling systems clone a fraction of packets or flows [14, 30, 29], sometimes along with internal switch statistics, from the forwarding engine to the switch CPU, which generates FRs that contain customizable metrics. The sampling reduces the CPU's workload, but in high speed networks aggressive sampling is necessary. For example, Cisco recommends a sampling rate of 1:1000 for monitoring 10 Gb/s links [58]). This significantly reduces packet and flow coverage, which in turn limits application correctness and accuracy. Despite its drawbacks, sampling is widely used in practice because it is available on many commodity switches.

Netflow ASICs. Some switches use custom hardware to generate FRs [59, 25, 60]. Depending on the design, the hardware may account for all packets [36], or packets

sampled at a low ratio [61]. Solutions that account for every packet typically use application specific integrated circuits (ASICs) that implement metric calculation in hardware and store per-flow state in high speed memory (e.g., TCAM). Older ASICs were limited to simple metrics, e.g., byte and packet counters [59]. Newer ASICs, such as those in switches designed for the Cisco Tetration platform [36], support additional metrics including latency, TCP window size, packet size, TTL, and TCP option variation [36]. The main drawbacks of Netflow ASICs are limited flow coverage, due to the necessarily small on-chip memories, and no programmability, as flow metrics are defined at fabrication time.

PFE Based Telemetry. An important recent trend is the rise of programmable forwarding engines (PFEs)[19, 62, 63] in next generation switches, line cards, and network interfaces. PFEs are emerging now because the chip area and power cost of programmability is becoming negligible, while the ever increasing number of protocols is making fixed-function devices less practical [64]. PFEs are appealing for telemetry because they can support custom packet processing at line rate, including the calculation of flow metrics. However, the few recent telemetry systems that leverage PFEs for FR generation force users to choose between throughput and coverage.

In FlowRadar [15], the PFE encodes flow keys and (optionally) metrics into counting Bloom filters [65], which a server later decodes and converts into FRs. Flow coverage is limited because overall system throughput is limited by the high cost of decoding, which increases exponentially with the number of flows being monitored.

Marple [11] is a system for streaming queries of network performance statistics, including queries for FRs. Marple splits the query processing between PFEs and a scale out key-value store, e.g., Redis [66]. The PFE partially computes the statistics requested by the query and streams updates to the key-value store, which aggregates the updates together. For efficient generation, the fields in the query are limited statistics and metadata that are *efficiently mergeable* [11], i.e., each update only requires the PFE to send a bounded amount of state to the backing store. Flow

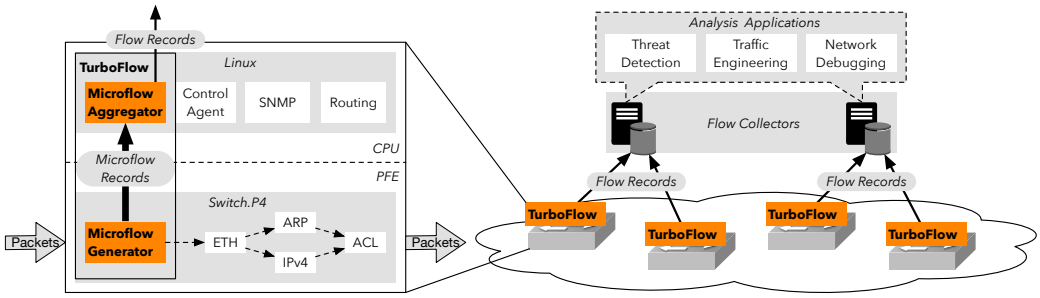

Figure 2.1: Deployment model of `TurboFlow`.

coverage is limited for Marple because it relies on scale out key-value stores (e.g., Redis) to track any flow state that does not fit into the PFE's small memory. For example, Marple is reported to require around 1 key-value server to service queries for a single 64 x 10 GbE switch [11].

2.3 TurboFlow Overview

`TurboFlow` is a FR generator for commodity programmable switches with P4 PFEs [19, 44]. It support high throughput, coverage, and programmability to enable network-wide monitoring at low cost. The key to achieving these goals is overcoming the memory-related bottlenecks that limit throughput in other high coverage telemetry systems. `TurboFlow` achieves this by exploiting heterogeneous hardware in the switch's PFE and CPU, along with common properties of telemetry workloads.

Figure 2.1 shows how `TurboFlow` integrates into a network infrastructure. The ingress pipeline of a switch PFE includes a `TurboFlow` cache that extracts telemetry-relevant features from packets and calculates/caches metrics for recently active flows. The cache regularly evicts partial records of flows (microflow records or mFRs) to a `TurboFlow` processor running on the switch's embedded management server, which aggregates the mFRs into complete FRs. Users can program `TurboFlow` to include custom metrics in FRs, constrained by the same requirement of efficient merge-ability as prior PFE-based telemetry systems [11].

Challenges. The high level challenge is overcoming bottlenecks on the path between a switch's PFE, where packets enter from data plane facing ports, and its main memory, where flow records are ultimately collected.

Figure 2.2 illustrates the general architecture of a PFE-based switch and the path of telemetry data. Packets enter through data plane ports; in 2019 switches typically have 32 or 64 100 Gb/s ports that can each be broken out into 10 or 25 Gb/s links. The data plane ports are wired to the PFE, which processes the packets at line rate. The PFE connects to an integrated management server via a PCIe bus and a DMA engine. The management server has all the same components as a standard server, e.g., an x86 CPU with 4 cores, 8GB of DRAM, and a persistent solid state memory.

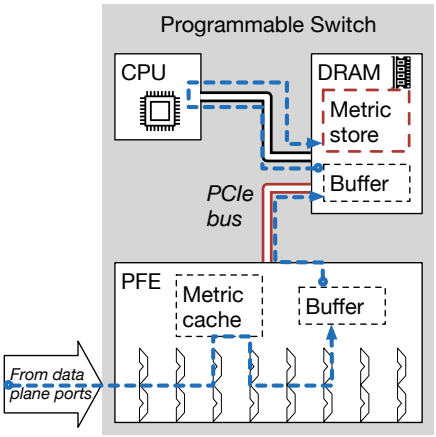

Figure 2.2: General architecture of a programmable switch. Dashed boxes show components of `TurboFlow`, dashed lines show movement of telemetry data from PFE to the metric table that tracks flow records. Red outlines indicate bottlenecks.

There are two main bottlenecks, outlined in red in Figure 2.2. To understand these bottlenecks, consider a concrete "terabit Internet measurement" scenario: a switch with a 3.2 Tb/s PFE, 4 CPU cores, and a 50 Gb/s PCIe link between its PFE and CPU. The switch connects 100 10 Gb/s links and needs to measure every packet that it forwards. This is a measurement workload of around 50 million packets per second. [1]

The first bottleneck is the limited bandwidth of the link between the PFE and main memory. It is around 2 orders of magnitude lower than the PFE (around 50 Gb/s versus $>$ 3.2 Tb/s). This means that a telemetry system can only afford to send a few bytes of data to the CPU for each packet. In the example scenario, the

[1] Based on an average of 500,000 packets per second per 10 Gb/s Internet link [67].

switch can move up to 125B per packet from PFE to main memory. Higher packet rates, additional links, and PCIe firmware or driver overheads can bring this down significantly [26].

The second bottleneck is the latency of accessing main memory from the CPU to find and update an entry in the metric table. Latency is high because main memory is large and off-chip. Fetching data from main memory takes orders of magnitude longer than fetching data from a CPU cache or performing computation (e.g., 100ns for a memory operation on a modern server versus >5ns for a L1 cache hit or >1ns for a SIMD operation).

As Chapter 4 finds, it is the typical bottleneck for CPU-based aggregation. For example, in the "terabit Internet measurement" scenario, a 100ns memory latency would limit maximum measurement throughput to around 10 million packets per second per core. This is 5X lower than required to measure the example scenario's 1 Tb/s Internet traffic workload with 50M packets per second.

Design. `TurboFlow` overcomes the bottlenecks of I/O bandwidth and memory latency by exploiting hardware heterogeneity in the switch and properties of common telemetry workloads. Figure 2.3 shows the general design of `TurboFlow`. Data follows a unidirectional flow through two main system components.

- A *Microflow generator* (mFR generator) in the PFE, implemented at line rate using the PFE's specialized compute and memory primitives, extracts features from each packet and uses a small cache to compute metrics summarizing recently active flows over short timescales. Whenever the mFR generator needs to make room for a new flow, it evicts a prior entry to a buffer in the switch's main memory.

- A *Microflow aggregator* (mFR aggregator), running on the switch management server, stitches the mFRs into complete flow records using a hash table optimized to leverage the CPU's cache and vector processing hardware.

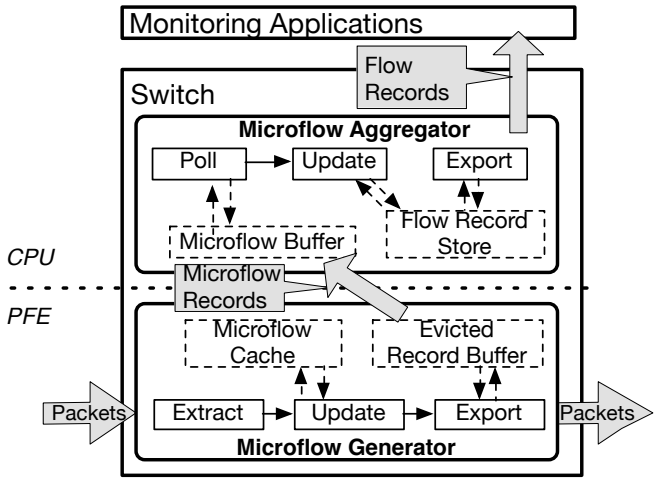

Figure 2.3: `TurboFlow` architecture.

The mFR generator eliminates the I/O bottleneck and helps alleviate the memory latency bottleneck. It eliminates the I/O bottleneck because a mFR is much smaller than an individual packet and summarizes multiple packets. For example, a mFR describing 8 32-bit metrics of TCP flows would be around 45B (13B for a TCP flow key and 32B for features). Even if each mFR only summarized 1 packet, the size reduction alone would often be enough to prevent I/O from being the bottleneck. For example, in the "terabit Internet measurement" scenario described above, the switch CPU had a budget of around 125B per packet.

In practice, mFRs summarize multiple packets, which further reduces the bandwidth required. The magnitude of reduction depends on the hit rate of the mFR's cache: each time a hit occurs, an additional packet is summarized into an existing mFR; each time a miss occurs, a mFR is generated. In practice, hit rates are high because of strong temporal locality, which Chapter 4.1 describes.

By summarizing multiple packets, mFRs also amortize the cost of memory operations, which are the bottleneck for CPU-based measurement (Chapter 4.2). For a CPU aggregator, most memory operations occur while looking up prior flow state. The number of memory operations per lookup is the same regardless of whether the CPU is processing packets or mFRs that each summarize an arbitrary number of

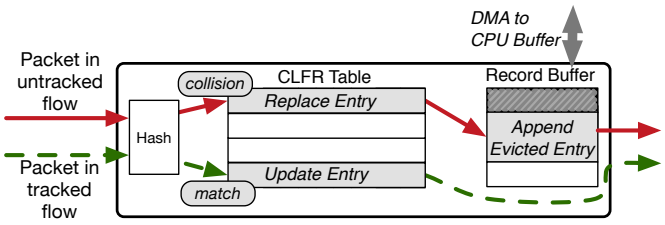

Figure 2.4: MFR generator data structure.

packets. Thus mFRs effectively multiple packet-rate measurement throughput by the average number of packets per mFR.

For example, in the "terabit Internet measurement" scenario, the switch was limited to aggregating 10M packets per second per core by a 100ns memory latency. In Internet scenarios, mFRs summarize an average of 3-10 packets each (Section 2.7). By processing mFRs instead of packets, the switch could measure 3X-10X more packets per second and cover its 50M packets per second workload using only 1 or 2 cores.

The mFR generator helps alleviate I/O and memory latency bottlenecks, but optimization of the mFR aggregator's back-end data structure is also important. For example, if each lookup required 10 100ns memory operations, the "terabit Internet measurement" switch would not be able to meet its throughput requirement, even using mFRs. To further reduce the cost of lookups (and thus maximize throughput), the mFR aggregator uses a hash table optimized to exploit CPU caches, wide registers, and prefetching to avoid unnecessary main memory operations.

2.4 Microflow Record Generation

The mFR generator produces mFRs that summarize burst of packets within flows. mFRs have the same format as FRs, depicted in Table 2.1. They can include any custom metrics that the PFE can compute. Capabilities depend on hardware. The most limited PFEs can perform logical operations, simple arithmetic, and limited-precision floating-point operations [52].

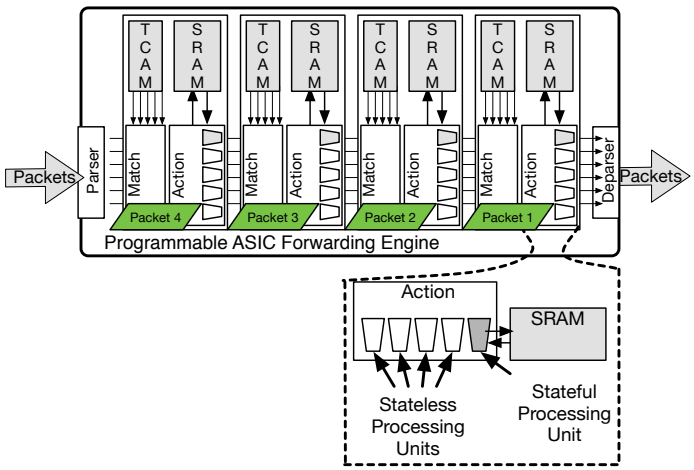

Figure 2.5: Generalized architecture of a P4 programmable ASIC.

Digest metrics are not necessarily the same as flow metrics – the back-end mFR aggregator applies a merge function that derives flow metrics from digest metrics. As described in Section 2.2, this limits flow metrics to decomposable functions [53]. Any associative and commutative function is trivially decomposable. The class includes e.g., the average, minimum, and maximum of any packet header field or metadata associated with processing, all of the metrics listed in Tables 2.1 and 2.2, and all statistics supported by Netflow ASICs.

The mFR generator, illustrated in Figure 2.4, contains a mFR table and a mFR record buffer. The *mFR table* maps packet keys to records based on their hash values. If the stored record has the same key as the packet, its metrics are updated. If the keys do not match, the stored record is copied to an *evicted record buffer* that is DMAed to the switch's main memory. In the table, the stored record is then replaced with a new record for the packet's flow.

The mFR generator is designed specifically so that it can map to the restrictive computational models of real PFE hardware. Below, we describe how it maps to programmable P4 ASICs [17, 18], which are highly restrictive; and NPUs [44, 68], which are less restrictive but have lower throughput.

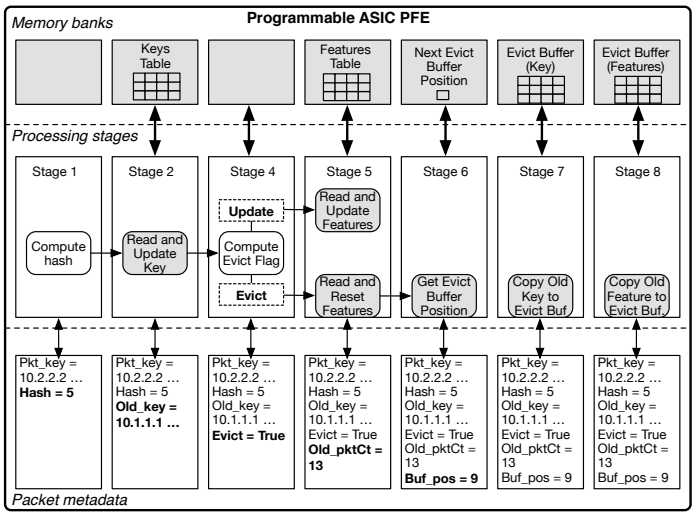

Figure 2.6: MFR generator mapped to a programmable ASIC.

Mapping to P4 Programmable ASICs. Figure 2.5 illustrates the general architecture of a P4 programmable ASIC. It has a pipeline of stages that each spends a fixed number of cycles processing each packet. A stage contains TCAM to store forwarding rules, SRAM for counters and other state that persists across packets, i.e., *register arrays* in P4 [69], a vector of processing units to modify packet headers or metadata carried along with it, and a small number of stateful processing units that can execute simple stateful programs while accessing the SRAM.

The architecture has two important benefits. First, it provides an extremely high and guaranteed throughput – any P4 program that compiles to the PFE will run at line rate, which is around 1 billion packets per second, or 1 packet per clock cycle, for recent designs [18, 17, 19]. Second, it is straightforward to implement functionality that can be expressed as match + action packet processing because the primitives of P4 map directly to the hardware.

Since the architecture is so specialized for match + action processing, it can be difficult to map more complex functions to the hardware. Sequential operations must be implemented as a sequence of actions in multiple stages. Stateful operations, which `TurboFlow` relies on, are highly constrained. In current ASICs, each register array can only be accessed once per packet, at a single location, to meet the per-stage time

budget. Stateful processing units enable programmable atomic updates to the register arrays, e.g., with simultaneous reads and writes or predicated updates. However, the atomic operations must be simple to meet chip space and timing budgets [17].

Figure 2.6 illustrates how the mFR generator maps to the processing stages in a programmable ASIC. It uses register arrays in SRAM banks to store the mFR table and evicted mFR buffer, packet metadata as a scratchpad for decision logic, and P4 tables to define control flow. The pipeline is unidirectional and operates on packets in parallel with other forwarding functionality. It is feed-forward and acyclic – there are no back branches or loops in the pipeline. This is a requirement of the P4 language and a constraint of today's reconfigurable ASICs.

Also, not depicted in the figure, all the persistent state is striped across memory banks. For example, an array of N 13 byte IP 5-tuple keys is actually 4 register arrays: 3 32-bit arrays and 1 8-bit array, each with length N and stored in a separate bank. This allows the pipeline to only need at most 1 read or write to any memory bank, per packet. The logical stages in Figure 2.6 implement the following logic.

1. Computes the hash of the packet's key.

2. Loads the key of the last flow with that hash from the *key table* into metadata, then writes the current packet's key back.

3. Sets a metadata flag indicating whether or not an evict is needed by comparing the current key with the previous key, which is now in metadata.

4. Loads the metrics values of the flow from the metric table into metadata, and resets or updates the metrics depending on the evict flag.

5. Loads the next free position in the output buffer, writes back a conditionally updated value: if the evict flag is set, the previous position plus the length of a mFR record; if not, the original value.

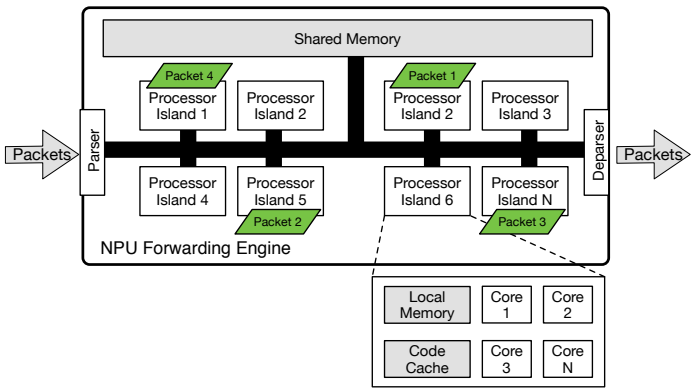

Figure 2.7: Generalized architecture of a NPU.

6. (through 8) Writes the key and metrics of the previous flow to the evicted buffers.

Mapping to an NPU. Network processors (NPUs) [63, 68] are an older and more flexible architecture for high throughput packet processing. However, they also have lower maximum throughput than programmable ASICs, fewer compiler guarantees on performance, and are less energy efficient. There are many NPU architectures, the most flexible of which use a pool of RISC cores to process packets, as Figure 2.7 depicts. The cores are arranged into islands with local SRAM for code and data, a large shared off-chip DRAM for forwarding tables and other large objects, and a high bandwidth switch fabric interconnecting the components.

Unlike programmable P4 ASICs, P4 instructions do not map directly to the hardware primitives of NPUs. Instead, NPUs support P4 with software libraries and toolchains [63] that compile P4 into routines for lower level languages, e.g., micro-C.

Mapping the mFR generator to an NPU requires optimization to maximize throughput. One consideration is minimizing the number of cycles required to process each packet. As Figure 2.4 shows, we found that some P4 primitives, most importantly applying tables and branching in the outer control flow of the pipeline, can be expensive on NPUs. We optimized the mFR generator to minimize the number of tables

```
// Metadata.
metadata tempMfr_t tempMfr;
metadata pktMeta_t md;

// Register arrays to store mFR table and evict buffer.
register keyArr[NUM_MICROFLOWS_TRACKED];
register pktCtArr[NUM_MICROFLOWS_TRACKED];
register evictBufArr[1];
register evictBufKey[BUF_SIZE];
register evictBufPktCt[BUF_SIZE];

// Control function -- call from P4 ingress.
control MfrGenerator {
   apply(UpdateKey);
   if (md.keyXor == 0) {
      apply(UpdateMetrics);
   } else {
      apply(ResetMetrics);
   }
}

// Tables.
table UpdateKey { default_action :UpdateKeyAction(); }
table UpdateMetrics { default_action :UpdateMetricsAction(); }
table ResetMetrics { default_action :ResetMetricsAction(); }

// Actions.
// Update key for every packet.
action UpdateKeyAction() {
   modify_field_with_hash_based_offset(md.hash, 0, key_field_list, HASH_SIZE);
   register_read(tempMfr.key, keyArr, md.hash);
   register_write(keyArr, md.hash, pkt.key);
   modify_field(tempMfr.keyXor,
      (pkt.key string^ tempMfr.key));
}
// Update metrics when there is no collision.
action UpdateMetricsAction() {
   register_read(tempMfr.pktCt, pktCtArr, md.hash);
   register_write(pktCtArr, md.hash, tempMfr.pktCt+1);
}
// Reset metrics and evict on collision.
action ResetMetricsAction() {
   register_read(tempMfr.pktCt, pktCtArr, md.hash);
   register_write(pktCtArr, md.hash, 1);
   register_read(tempMfr.evictBufPos, evictBufArr, 0);
   register_write(evictBufArr, 0, tempMfr.evictBufPos+1);
   register_write(evictBufKey, tempMfr.evictBufPos, tempMfr.key);
   register_write(evictBufPktCt, tempMfr.evictBufPos, tempMfr.pktCt);
}
```

Figure 2.8: mFR generator psuedocode for NFP-4000.

Added Instruction	Example P4 14 Code	Throughput Change
Control flow branch	`if (...){ apply(...); }`	-15.5%
Apply table	`apply(...);`	-9.5%
Read memory	`register_read(...);`	-3.0%
Write memory	`register_write(...);`	-3.0%
Modify header	`modify_field(...);`	-0.5%

Table 2.4: P4 primitives cost on the NFP-4000.

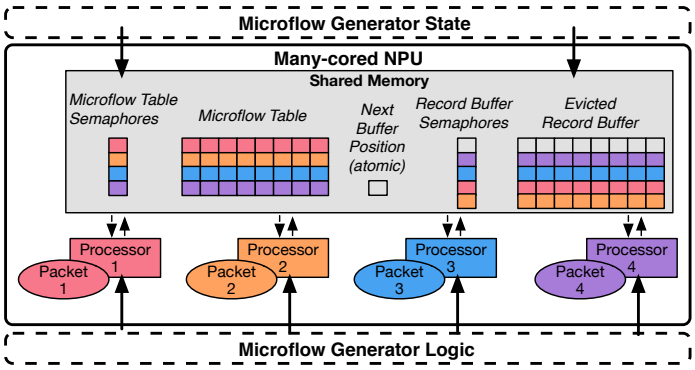

Figure 2.9: Mapping the mFR generator to a NPU with fine-grained semaphores.

and branches in the outer control flow. Figure 2.8 shows P4-14 psuedocode after optimization, which only requires applying 2 tables per packet.

A second performance concern is synchronizing state without contention. Each core operates on a different packet concurrently and needs thread safe access to the shared mFR state. The P4 library for the NPU that we targeted, the NFP-4000 [63], has a coarse-grained semaphore that could only lock the entire mFR table. To eliminate contention, we implemented a simple, spin-lock semaphore that allows a core to lock a single row of the mFR table or record buffer, as shown in Figure 2.9.

2.5 Microflow Record Aggregation

The mFR aggregator runs on the switch CPU and stitches mFRs, streamed up from the PFE, into full FRs. The challenge in designing the mFR aggregator is optimizing

the key-value data structure to maximize the rate at which it can map mFRs to complete FRs.

Reading mFRs

The DMA engine of the PFE copies mFRs from the evicted record buffer into free cells in a larger ring buffer in main memory. The mFR aggregator processes the mFRs in batches and sends the addresses of free cells back to the DMA engine. This design is similar to high performance NIC drivers [70], and allows the DMA engine to dynamically vary the copy rate, as long as there are free cells available.

To use multiple cores, `TurboFlow` spawns multiple mFR aggregators that each maintain their own buffers. The PFE statically load balances mFRs across the buffers based on key.

Aggregating mFRs

The aggregator stores FRs for active flows in a hash table. For each mFR, the aggregator either updates the metrics of an existing FR, inserts a new FR, or, for `TCP` packets with `FIN` flags, copies a FR to an output buffer, and removes it from the hash table. The hash table is the core bottleneck for the aggregator, and optimizing it was the focus of our implementation. We found four optimizations that significantly improved performance.

Linear Probing. `TurboFlow` uses a linear probing hash table [71]. Linear probing has high cache locality, which significantly increases throughput given the small size of individual FRs. When a hash table miss occurs, the next FR to check will likely be in the CPU's cache already.

Flat Tables. The aggregator stores FRs directly in the hash table, i.e., each slot stores a FR, rather than a pointer to a container. This eliminates dereferencing,

which saves cycles and further reduces cache misses.

128 Bit Integer Keys. The aggregator represents flow keys as two 64 bit integers: the first stores IP addresses; the second stores ports, protocol, and (optionally) physical link ID. This allows the key comparison function to use SSE 4.1 operations to compare a pair of 128 bit keys in only 2 instructions [72].

Lookup Prefetching. The aggregator batches lookups to mask memory latency. It prefetches the hash table slots where each FR in the batch is most likely to be before processing, so the memory lookups occur in parallel with the processing of the first few records. Prefetching also compliments linear probing: when a record is *not* in the expected slot, the linear probing algorithm is likely to have placed it in the slot immediately proceeding, which would also be loaded by the prefetch.

Exporting Flow Records

A separate thread of the aggregator packs the evicted FRs into packets and exports them to collection or analysis servers. It also periodically scans the hash table entries and expires all flows that have been inactive for longer than a pre-configured length of time.

2.6 Expected Worst Case Performance

Allocating more PFE memory to the mFR generator reduces the rate of mFRs sent to the CPU by decreasing collisions, which lets `TurboFlow` scale to traffic higher rates and leaves more CPU cycles for other applications. But how much PFE memory should an operator allocate to meet a target mFR rate?

To guide configuration, we derive Equation 2.2, an expected worst case bound for the rate of the mFR stream based on the size of the mFR table in the PFE (T), expected packet rate ($E[p]$), flow rate ($E[f]$), and number of simultaneously active

flows ($\hat{a}$). Table 2.5 lists the expected worst-case rates in terms of $\hat{a}$, given a fixed packet rate and flow rate.

Table Size	a	$2 \times a$	$3 \times a$	$4 \times a$	$5 \times a$
$P[eviction]$	.65	.40	.28	.22	.18
Packets : mFR	1.53	2.5	3.57	4.54	5.55

Table 2.5: Eviction chance with a active flows.

The expected worst-case rate depends on the probability that an individual packet causes an eviction, which Equation 2.1 describes. An Eviction occurs when there is a collision. The probability of a packet colliding with a prior entry is equal to 1 minus the probability that all other active flows map to *different* slots than the packet. $\frac{1}{T}$ is the probability of a single flow having the same hash value as the packet, $(1 - \frac{1}{T})$ is the probability of a single flow having a *different* hash value, and $(1 - \frac{1}{T})^{\hat{a}})$ is the probability that all $\hat{a}$ active flows have different hash values than the current packet.

Chapter 4 derives a more complete model for the performance of an entire telemetry system and demonstrates its effectiveness in a range of workloads.

$$P[eviction] = 1 - (1 - \frac{1}{T})^{\hat{a}}) \tag{2.1}$$

$$E[m] = E[f] + (E[p] - E[f]) * P[eviction] \tag{2.2}$$

2.7 Evaluation

We implemented `TurboFlow` and used benchmarks, simulation, and analysis to answer the following questions:

- What are the computational and memory requirements for `TurboFlow`?
- How much do optimizations in the mFR aggregator improve throughput?

Trace	Capacity	Flow Rate	Packet Rate
Internet Router Link	10 Gb/s	5K - 40K	0.250 M - 0.6M
DC ToR Switch	1280 Gb/s	478K	124.0M
DC Agg. Switch	1440 Gb/s	1291K	134.0M

Table 2.6: Evaluation workloads.

- What is the overall monitoring capacity of a switch running `TurboFlow`, and how difficult is it to tune?
- What is the infrastructure cost of high coverage monitoring with `TurboFlow`?

The performance modeling framework in Chapter 4 generalizes the results from this evaluation section.

Experimental Setup

Evaluation Platforms. We implemented the programmable ASIC and NPU designs of TurboFlow. [2] The programmable ASIC implementation targeted the Wedge 100BF-32X [38], a 32x100 GbE switch with a Tofino [19] PFE and an Intel D1517 quad core CPU with 8 GB of RAM. The NPU implementation targeted a switch built using a commodity server with a 4x10 GbE NFP-4000 [44] NPU, packaged as a PCIe card, and an AMD Opteron-6272 CPU.

Each implementation had the same mFR aggregator, written in C++, but different mFR generation code, written in a combination of P4 and platform-specific languages, for access to hardware features not supported by P4. The Tofino implementation required platform-specific code for the stateful operations, while the Netronome implementation used our custom semaphore, written in micro-C.

[2] `TurboFlow` code repository: `https://github.com/jsonch/turboflow`

Benchmark Workloads

We configured `TurboFlow` to produce FRs that included IP 5-tuples and 4 metrics: packet count, byte count, start timestamp, and end timestamp. We benchmarked it with traces that represent Internet router and data center switch workloads, as Table 2.6 summarizes.

Internet Routers. We used 8, 1-hour long traces from 10 Gb/s links between core Internet routers [67], collected in 2015. Each trace contains 1 - 2 billion anonymized packet headers, representing over 99% of the packets that crossed the links during the collection periods. To scale the workload up to Tb/s rates, we modeled a router that monitors many 10 Gb/s links with independent traffic flows. In `TurboFlow`, we allocated a different segment of the mFR table for each link, and statically load balance mFRs from each link to the CPU buffers. This represented a scenario where the packet rate, flow rate, and number of active flows, i.e., all the variables in the worst-case performance equation, scaled linearly with link capacity.

Data Center Switches. We generated packet traces of a simulated data center using YAPS [73], an event based simulator parameterized by the data center traffic statistics reported in [35]. YAPS is based on the simulators used in other recent work [74, 75]. We modeled a 40 GbE two-tier data center network composed of 144 *end hosts* that generated traffic, 9 *ToR switches* that connected to end hosts, and 4 *aggregation switches* that interconnected the ToR switches.

Component Performance

First, we measure the performance of the mFR generator and mFR aggregator to understand the overall throughput of `TurboFlow`.

mFR to Packet Ratio. The mFR aggregator reduces the switch CPU's workload by summarizing multiple packets into each mFRs. We quantify the workload reduc-

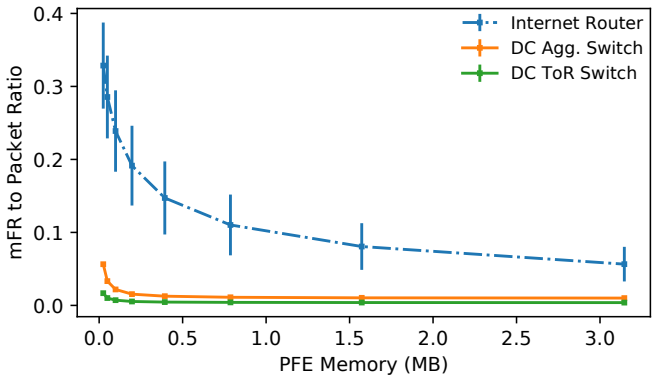

Figure 2.10: PFE memory vs. ratio of mFRs to packets.

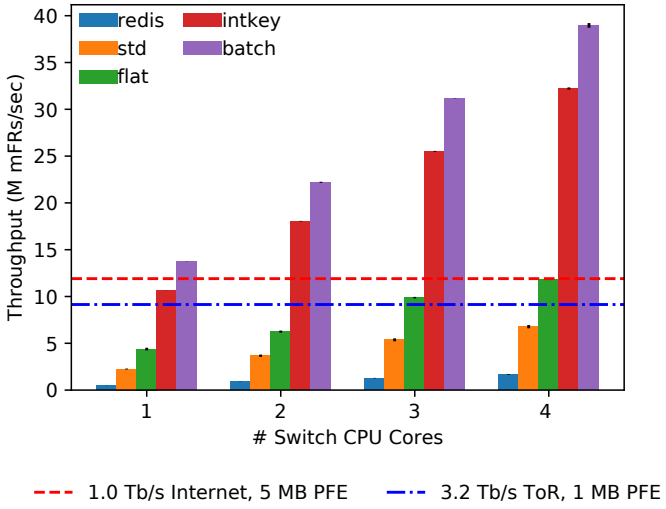

Figure 2.11: mFR aggregator throughput with optimizations.

tion in terms of the *mFR to packet ratios*. Figure 2.10 plots mFR to packet ratios for the Internet router and DC switch traces. A small amount of PFE memory, e.g., around 1 MB, reduces CPU workloads by a factor of at least 10 in all traces, with the reduction being more significant in data center traces. There, 100 KB of PFE memory reduced the CPU workload by a factor of 45 (for aggregation switches) and 135 (for the top of rack switches). As Chapter 4 explains, telemetry systems gain significant benefit from a small cache because of inherent locality in their workload.

Switch CPU Throughput. Figure 2.11 plots throughput of the mFR aggregator with different optimizations, averaged over 100 trials. The rightmost bar of each cluster (*batch*) shows throughput with all the optimizations described in Section 2.5. The mFR aggregator had an average throughput of 13.73M mFRs/s with 1 core, and scaled almost linearly with additional cores; cores 2 through 4 increased throughput by 8.4M, 8.9M, and 7.79M mFRs/s, respectively.

The leftmost bar (*redis*) in Figure 2.11 shows a baseline `TurboFlow` aggregator implemented using Redis [66]. It scaled well, but was much less efficient than `TurboFlow`. *The difference is a factor of 20.* The Redis implementation is a strawman that emphasizes the benefit of optimization when constrained to the switch CPU. However, even compared to a much more efficient C++ implementation using a `std::unordered_map` (*std*), the `TurboFlow` optimizations still provided a 5.67X throughput increase.

The horizontal lines in Figure 2.11 illustrate *why* the extra throughput matters, with two workloads that require the switch CPU to process around 10M mFRs/s. The red line illustrates the required throughput to monitor 1 Tb/s of traffic with the multi-link Internet router workload with and 5 MB of PFE memory dedicated to `TurboFlow`; the blue line illustrates requirements for 3.2 Tb/s of DC traffic at a ToR switch (derived by time-accelerating the 1.28 Tb/s ToR trace by a factor of 2.5). The mFR rates for these workloads are computed based on the packet rates of the traces and the packet to mFR ratios shown in Figure 2.10. The `TurboFlow` aggregator can support both of these workloads with 1 or 2 cores, which neither the Redis nor C++ baselines could support, even using all 4 cores.

Chapter 4 generalizes these results by showing how performance of the CPU-based mFR aggregator varies with workload and measurement tasks.

Microbenchmarks

Microbenchmarks measure the compute, memory, and bandwidth requirements of `TurboFlow` components, and show how they change with the of additional metrics.

	Logical Stage	# Tables	# VLIWs	# SALUs	# TCAMs
1.	Compute hash	0	0	0	0
2.	Update key	4	3	4	0
3.	Set evict flag	1	1	0	0
4.	Update metrics	4	3	4	12
5.	Load buffer pos	1	2	1	3
6&7.	Update evicted buf.	8	2	8	0
	Total	9.38%	2.86%	35.42%	5.21%

Table 2.7: Tofino pipeline usage for `TurboFlow`.

Tofino PFE. On the Tofino, the mFR generator was compiler guaranteed to run at line rate. The primary questions was how many of the Tofino's computational resources `TurboFlow` required, which determines how much room there is for other functionality. Table 2.7 shows requirements for three important resources, based on output from the Tofino compiler. The mFR generator required no more than 36% of any resource, which leaves room for many other functions to process packets in parallel with `TurboFlow`.

`TurboFlow` used under 10% of the table, VLIW, and TCAM resources, which common data plane functions such as forwarding and access control rely on.

`TurboFlow` consumed a larger portion of the Tofino's stateful ALUs (SALUs). Recent prototype data plane applications would also use SALUs [76, 77, 11, 15]. Although not all of these applications may be able to map to the Tofino, we can estimate an upper bound for the number of SALUs they would require by counting the number of register reads or writes in their P4 code. By this metric, we estimated that a data plane cache for read heavy key-value stores would require 7 SALUs [76], a Paxos implementation [77] would require 9, and a simple EWMA estimate of link utilization for traffic load balancing [27] could be implemented with 1. Based on SALU requirements, all of these applications could be deployed concurrently with `TurboFlow`.

Table 2.7 also shows that the hash computation required no additional resources.

Cycles	Mem Ops.	Hashing	Apply Tables
3423	66.76%	2.13 %	27.81%

Table 2.8: Single thread cycle count on NFP-4000.

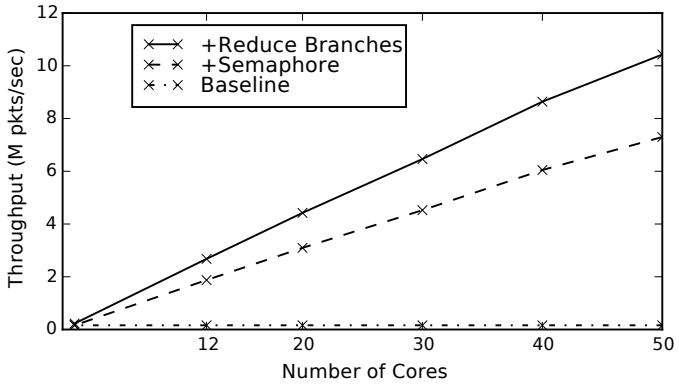

Figure 2.12: Packet rate throughput on the NFP-4000.

We avoided the need to compute it by configuring the SALUs to compute the hash of the packet's key while accessing the register arrays.

NFP-4000 PFE. Figure 2.12 plots the performance of `TurboFlow` on the NFP-4000. With all cores of the device enabled (running with 8 threads per core) and the optimizations from Section 2.4, it sustained around 11 M packets per second, enough to saturate the 40 Gb/s interface even with small 300B packets.

Table 2.8 breaks down the processing time of a single invocation using the NFP-4000's single thread debugger. We measured an average per-packet cycle count of 3423 for the `TurboFlow` mFR generator. As Table 2.8 shows, memory accesses dominated the cost, which took approximately 200 cycles per read or write.

This cost is partially masked by the NFP-4000's threading model. When one thread pauses to access memory, the core switches to another thread and performs computation. This could reduce the effective cost of a 200 cycle memory operation to $200/8 = 25$ cycles, in a compute heavy program.

Resource	4 Metrics	8 Metrics	Rel. Cost
(Tofino) Tables	18/64	26/64	44%
(Tofino) sALUs	17/44	25/44	47%
(NFP-4000) Cycles	3423	4415	29%
(Switch CPU) mFR Throughput	13.73M	12.57M	8.4%

Table 2.9: The cost of generating additional metrics.

The effect is more modest for the `TurboFlow` mFR generator. Accounting for code outside the scope of P4, the single thread debugger reports a processing time of 6027 cycles per packet. This corresponds to a single-thread throughput of around 200K packets per second. When the NFP-4000 (in debug mode) runs the `TurboFlow` mFR generator on a single core with all 8 threads, it achieves a throughput of around 300K packets per second, roughly suggesting a 50% throughput improvement due to latency masking from 8 threads.

Running additional data plane functions alongside `TurboFlow` would reduce throughput. To estimate how much, we measured the cost of all P4 primitives. The most expensive operations were applying tables (around 1200 cycles each) and reading / writing to register arrays (around 200 cycles each). Based on these measurements, we estimate that running `TurboFlow` along with 5 custom forwarding tables and another stateful P4 program that requires 9 register reads and 9 register writes, e.g., a Paxos [77] implementation, would cost around 13,000 cycles per packet, resulting in a throughput of around 3 M packets per second, or 20 - 30 Gb/s with average packet sizes around 600 B - 1 KB, which is common [67].

Additional Metrics. Table 2.9 shows the cost of generating FRs with 4 additional metrics (maximum queue depth, packet size, inter-arrival time, and average queue depth).

For the switch CPU, the additional metrics have limited impact because the main bottlenecks were per-mFR operations rather than per-byte operations.

PFE Memory	Internet Link	ToR Switch	Aggregation Switch
PCIe load (mFRs to CPU)			
49 KB	22.57 Mb/s	242.93 Mb/s	856.99 Mb/s
98 KB	19.01 Mb/s	173.06 Mb/s	563.17 Mb/s
196 KB	15.39 Mb/s	129.52 Mb/s	397.63 Mb/s
786 KB	9.27 Mb/s	100.04 Mb/s	286.92 Mb/s
1572 KB	7.05 Mb/s	94.65 Mb/s	267.52 Mb/s
Network load (FRs to collector)			
-	1.75 Mb/s	89.71 Mb/s	247.92 Mb/s

Table 2.10: Communication overheads for `TurboFlow`.

On the Tofino, this requires 8 additional tables and sALUs to widen the mFR table and evicted mFR buffer. Since processing time is constant, this does not impact its throughput.

On the NFP-4000, the additional memory operations increases the per-packet processing time by 29%. Processing time is still low enough to maintain a 40 Gb/s line rate with 400B packets.

Ultimately, PFE resources will limit how many metrics a telemetry system can support concurrently. Chapter 3 generalizes `TurboFlow` to overcome this (and other) limitations, while Chapter 4 identifies the crossover points at which the general approach can also provide better performance.

PCIe and Network Overhead. Table 2.10 shows that even when only using small amounts of PFE memory, data rates to the CPU are low, in the sub 1 Gb/s range. For context, the PCIe 3.0 x4 interfaces of the Tofino and NFP-4000 have theoretical maximum throughputs of 32 Gb/s. Network overhead for exporting the complete FRs to collection servers is even lower, requiring less than $\frac{1}{1000}$ as much bandwidth as the original monitored traffic.

PFE Memory	Number of CPU Cores 1	2	3	4
Internet Router				
0 MB	300 Gb/s	600 Gb/s	900 Gb/s	1200 Gb/s
1 MB	600 Gb/s	1200 Gb/s	1700 Gb/s	2200 Gb/s
4 MB	800 Gb/s	1400 Gb/s	1900 Gb/s	2500 Gb/s
8 MB	900 Gb/s	1600 Gb/s	2300 Gb/s	2800 Gb/s
16 MB	1100 Gb/s	1800 Gb/s	2600 Gb/s	3200 Gb/s
24 MB	1200 Gb/s	2000 Gb/s	2800 Gb/s	3500 Gb/s
Data Center Aggregation Switch				
1 MB	6.4 Tb/s	9.3 Tb/s	13.6 Tb/s	14.4 Tb/s

Table 2.11: Aggregate monitoring capacity for `TurboFlow`.

Monitoring Capacity and Tuning

Using the benchmark results and statistics from the workload traces, we analyzed the effective monitoring capacity of a Wedge BF32-100X running `TurboFlow`, and the difficulty of tuning.

Aggregate Capacity. Table 2.11 summarizes the monitoring capacity of `TurboFlow` in terms of Tb/s of link capacity, using different amounts of PFE memory and numbers of switch CPU. We derived these capacities based on the average packet sizes in the workload traces, the mFR to packet ratios in Figure 2.10, and the average mFR throughputs in Figure 2.11. The table shows that `TurboFlow` can scale to produce FRs for terabit rate traffic with both Internet and data center workloads, using reasonable amounts of PFE memory and CPU cores.

Without *any* PFE processing, i.e., the PFE has no mFR table and sends every packet to the CPU as a mFR representing 1 packet, `TurboFlow` scales to 1.2 Tb/s of aggregate Internet links when using all 4 cores. A small amount of PFE memory, 1 MB, can replace 2 of those cores to reach the same capacity. At the other extreme, the switch can also scale to the same traffic rates using 24 MB of PFE memory and only 1 CPU core.

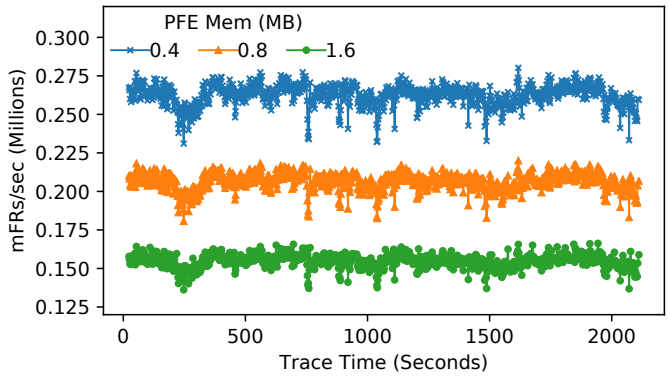

Figure 2.13: mFR rates over time in 1 core Internet trace.

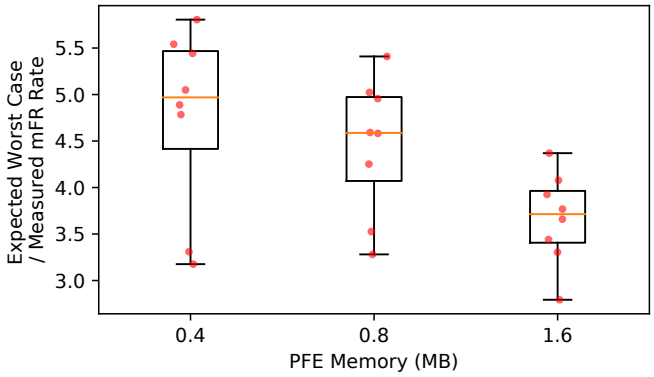

Figure 2.14: Worst case to measured mFR rates for 2015 Internet router traces.

For perspective, Marple [11], the only other recent PFE accelerated system capable of efficiently generating FRs with these metrics, is estimated to require an 8 core dedicated server to support a 64x10 GbE switch [11] with Internet scale workloads derived from the same links.

Tuning. To analyze the difficulty of tuning `TurboFlow`, i.e., selecting how much PFE memory to use, which determines mFR rate and thus CPU load, we measured the *stability* of configurations over time and the *safety* of the analytic formulas we derived in Section 2.6.

Figure 2.13 plots mFR rates during .5 second intervals in the 12/2015 core Internet router trace. MFR rates were stable throughout the duration of the trace and variance

was low. Given the stability, it would be practical to tune `TurboFlow` based on a short initial sample of traffic in these workloads.

Figure 2.14 plots a histogram of the ratio of worst-case expected mFR rate to average measured mFR rate, with trials for all 8 2015 core Internet router traces. The ratio is always above 1, demonstrating that the worst-case formula in Section 2.6 provided a safe bound in all scenarios. Since switch CPU load is stable over time, operators can use the formulas to find an initial configuration and later tune PFE memory allocation to meet target switch CPU loads.

Telemetry Infrastructure Cost

To analyze the cost of high coverage monitoring with `TurboFlow`, we modeled the equipment and power costs of a generating and analyzing FRs in a network built from Wedge 100BF-32X switches and commodity servers. We compared the cost of monitoring with `TurboFlow` to a model that represents a lower bound estimate of cost using either Marple [11] or FlowRadar [15].

Cost Model

The cost model calculates the *equipment cost* (in dollars) and *power consumption* (in Watts) of monitoring with respect to the capacity of the monitored links (in Tb/s). We assume that cost is continuous, e.g., it is possible to deploy a fraction of a server, and that cost scales linearly with the total capacity of the monitored links. Linear scaling is a reasonable assumption because monitoring work can be statically load balanced across the servers, e.g., based on link or traffic flow, and the overhead of transferring FRs across the network is negligible (Table 2.10).

$$Cost = \frac{FlowRate(w)/Capacity(w)}{ServerTput(t,i)/ServerCost} \tag{2.3}$$

Equation 2.3 is the cost function per Tb/s of traffic for a monitoring task t (either generating FRs or analyzing FRs), using a telemetry system i (either `TurboFlow` or

FlowRadar / Marple) with a traffic workload w (Internet router, DC ToR switch, or DC aggregation switch). The numerator describes the workload; it normalizes their flow rates by capacity for equal comparison. The denominator expresses the number of FRs that can be processed per cost unit (dollar or Watt), which depends on the task, the telemetry system, and the cost of the servers.

Raw Cost. We based server cost on a reference server with an Intel Silver 4110 CPU 8 core CPU, which listed for approximately $3500 at the time of this work (2017) and uses around 600 Watts under full load.

For perspective, we also reference the cost of the raw switches as a lower bound on the cost of a data plane with no monitoring. Wedge 100 series switches cost around $3600 per Tb/s in 2017 and have a typical power consumption of around 150W per Tb/s [78].

Server Throughput. We estimate per-core server throughput for FR generation and analysis.

We define *FR generation* work as any processing required to collect telemetry data from switches, convert it into FRs, and copy the records into in-memory buffers for analysis applications to consume. For `TurboFlow`, the processing is simply collecting FRs from switches and copying them to buffers for analysis. We use a conservative throughput of 50M FRs/s, which corresponds to filling the buffer with 25B FRs at a bit-rate of 10 Gb/s, well within the capacity of a single server core [70]. We factored in `TurboFlow`'s usage of the switch CPU by adding 7.8W per Tb/s to the power cost of FR generation, modeling a scenario where `TurboFlow` fully utilizes the entire 25W CPU to generate FRs for its 3.2 Tb/s of switch links.

Marple and FlowRadar do post processing to aggregate data from the switch into complete FRs. We estimate an upper bound on their throughput based on the optimistic assumption that the processing servers will only need to do 1 key-value operation per flow. In practice, this would be higher since both systems export

Workload	Switches	+ Generation	+ Analysis
Equipment Cost (per Tb/s)			
DC ToR	$3600	$3603 (+ 0.1%)	$3642 (+ 1.2%)
DC Agg.	$3600	$3608 (+ 0.2%)	$3702 (+ 2.9%)
Internet	$3600	$3636 (+ 1.0%)	$4059 (+ 12.8%)
Power Cost (per Tb/s)			
DC ToR	150 W	158 W (+ 5.6%)	164 W (+ 10.0%)
DC Agg.	150 W	159 W (+ 6.1%)	174 W (+ 16.7%)
Internet	150 W	163 W (+ 9.2%)	234 W (+ 56.3%)

Table 2.12: Cost of monitoring infrastructure with `TurboFlow`.

multiple records per FR. We measured the per-core throughput of Redis [66], a widely used scale out key-value store, at 625K updates/s per core. This is consistent with other benchmarks [66].

Analysis work includes any processing of FRs to extract higher level information. To estimate the throughput of a FR analysis process, we implemented a simple traffic classifier in C++ using Dlib [79]. The classifier predicts the application that generated a flow record (e.g., https, ssh, dns), using flow metrics described in prior work [80]. We benchmarked the per-core throughput of the classifier at 4.25 M FRs/s on the reference server.

Workload Profiles. We use the flow rates and capacities listed in Table 2.6 to normalize cost. For the Internet workload, where average flow rate depends on trace, we used the highest rate of 40K / s.

Cost Analysis

Table 2.12 summarizes the modeled cost of high coverage monitoring with `TurboFlow`. Generating FRs added under 1 % to equipment cost and under 10 % to power cost. Analyzing the FRs, which depends entirely on the analysis application rather than

Workload	**DC ToR**				**DC Aggregation**				**Core Internet**			
System	TurboFlow		PFE Accelerated		TurboFlow		PFE Accelerated		TurboFlow		PFE Accelerated	
Cost Metric *(Per Tb/s)*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*	*Unit*	*Power*
Generation	$3	8 W	$268	44 W	$8	9 W	$645	107 W	$36	13 W	$2880	479 W
Analysis	$39	6 W	$39	6 W	$94	15 W	$94	15 W	$423	70 W	$423	70 W
Total	$42	14 W	$308	51 W	$102	24 W	$740	123 W	$459	84 W	$3303	550 W
TurboFlow Saving	**$265, 36 W per Tb/s**				**$637, 98 W per Tb/s**				**$2844, 466 W per Tb/s**			

Table 2.13: Cost comparison between `TurboFlow` and other PFE accelerated telemetry systems.

the telemetry infrastructure, was more expensive. However, even though the traffic classifier that we benchmarked was unoptimized and did computationally expensive machine learning, the overall monitoring cost was still reasonable because of the highly efficient FR generation with `TurboFlow`.

Table 2.13 compares the cost of `TurboFlow` to other recent PFE accelerated telemetry systems. `TurboFlow` reduced both equipment and power costs of generating FRs by a factor of > 5 for all workloads. This corresponded to a cost reduction of > 3, even when also running the analysis application.

Chapter 3

Increasing Flexibility

3.1 Introduction

`TurboFlow` showed how a hybrid telemetry system can exploit recently emerging programmable forwarding engines (PFEs) to meet the performance requirements necessary for deployment in high speed networks. Performance matters, but there are also important flexibility requirements for practical deployment. This chapter addresses the question of leveraging PFEs for performance *while also meeting key flexibility design goals*.

Flexibility requirements arise from the wide range of applications that a telemetry system can potentially support. In a large network, it is likely that a telemetry system will need to support multiple monitoring applications. For example, an anomaly detector [42, 81, 5, 41], a load balancer [6, 82], and a network route optimizer [83, 84, 85]. As Chapters 1 and 2 described, these applications have different objectives and are likely to require different flow metrics. Further, some applications require late binding (e.g., interactive debuggers [11] that change metrics at runtime), or metrics that are complex to compute.

`TurboFlow` (and other PFE-accelerated telemetry systems [11, 15, 16]) do not support such flexibility requirements well. These systems obtain throughput by com-

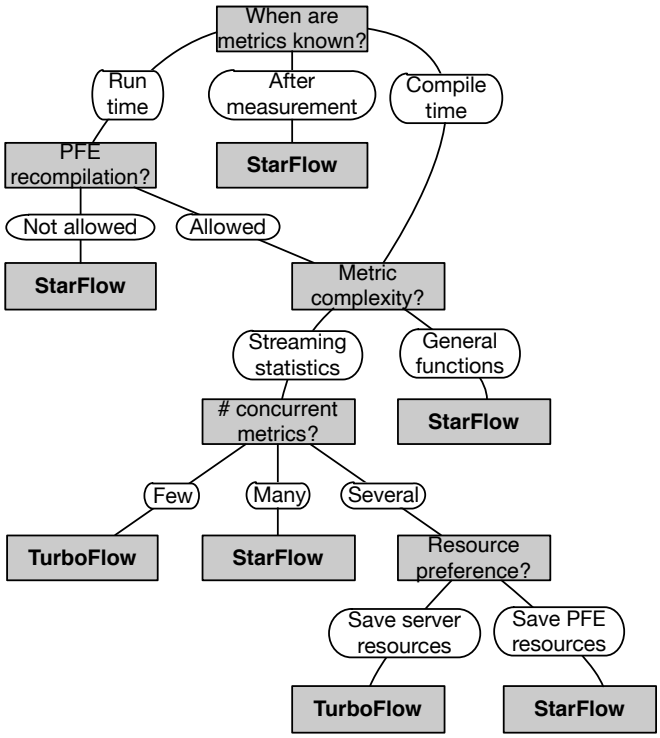

Figure 3.1: `TurboFlow` versus `*Flow`.

piling *metric queries* to the PFE, i.e., functions that calculate metrics for recently active flows. This comes with inherent limits to flexibility. Concurrent measurement and complex metrics are limited by the compute and memory resources of the PFE and its restrictive processing model, while dynamic applications are limited by the high cost of recompiling a PFE's program.

Introducing *Flow. We introduce `*Flow`, a hybrid telemetry system that generalizes the design of `TurboFlow` to support the flexibility goals of concurrency, late binding, and complex metrics in exchange for a limited reduction in performance. Figure 3.1 compares `TurboFlow` and `*Flow`, illustrating the scenarios in which one system is better than the other.

The core insight of `*Flow` is that flexibility is challenging for `TurboFlow` and other prior telemetry systems because they compile *too much* of a metric query to

the PFE. The issue can be resolved without significant impact to performance by carefully lifting parts of the query up to software running in a server (e.g., the mFR aggregator in `TurboFlow`).

At a high level, a metric query can be decomposed into three logical tasks: a *selection* task that extracts metric-relevant packet header features; a *grouping* task that maps packets to flows; and a *calculation* task that computes metrics over a stream of grouped packet features.

`*Flow` is based on the observation that PFEs and servers are best suited for different tasks. The primary benefit of using the PFE is for selection and grouping, which it can perform efficiently using its line rate compute and memory hardware. The challenge lies in implementing calculation, which is metric dependent and potentially complex.

For servers, the situation is exactly reversed. A server's limited I/O prevents it from implementing selection on a high throughput packet stream, while its high memory latency makes grouping slow. However, given a batch of packet features that all belong to the same flow, a server can calculate metrics or perform coarse grained re-grouping quickly and efficiently, as no per-packet flow lookups are required.

`*Flow`'s design, depicted in Figure 3.3, plays to the strengths of both PFEs and servers. Instead of compiling entire queries to the PFE, `*Flow` places parts of the *selection* and *grouping* tasks that are common to all queries in the PFE. The PFE exports a stream of digests that contain flow-batched packet features, which a server can compute a diverse range of metrics over *without having to do expensive selection or grouping*. This design maintains the performance benefits of using a PFE while eliminating the root causes of flexibility limitations.

Grouped Packet Vectors. To lift calculation off of the PFE without sacrificing performance, `*Flow` introduces a compact but flexible intermediate format for telemetry digests that stream from a PFE to a backend server: *grouped packet vectors* (GPVs). A GPV contains a flow key, e.g., IP 5-tuple, and a variable-length list

of packet feature tuples, e.g., timestamps and sizes, from a sequence of packets in that flow.

GPVs allow a hybrid telemetry system to achieve high performance because they overcome the key I/O and memory latency bottlenecks. They are compact, which eliminates the I/O bottleneck between the PFE and server in a hybrid system. GPVs also amortize flow lookup latency (which is high due to inherent memory latency) because all the data in a single GPV belongs to the same flow.

At the same time, GPVs enable significant flexibility in a hybrid telemetry system because a GPV stream contains packet features, which expand into a large set of metrics. The backend server can time multiplex its hardware to calculate subsets of those metrics concurrently, adjust metrics at runtime or after telemetry data has been collected, and leverage its general capabilities to support complex metrics.

Dynamic in-PFE Cache. Switches generate GPVs at line rate by compiling the `*Flow cache` to their PFEs alongside other data plane tasks. The cache is an append-only data structure that maps packets to GPVs and evicts them to software as needed.

An append-only cache is more complex than prior data plane caching systems [11, 76, 86] and is a challenge to implement in a PFE. Due to a PFE's inherent restrictions on stateful operations, common algorithms for memory management cannot be supported. To utilize the limited PFE memory, e.g., around 10MB as efficiently as possible, the `*Flow cache` approximates cache placement and introduces a new approximate dynamic allocation algorithm.

The resulting data structure exploits locality properties of a multi-flow workload to maximize the benefit of a small cache implemented on restrictive, line-rate hardware.

Implementation and Evaluation. We implemented the `*Flow` cache[1] for a 100BF-32X switch, a 3.2 Tb/s switch with a Barefoot Tofino [19] PFE that is programmable with P4 [69]. The cache is compiler-guaranteed to run at line rate and uses a fixed

[1] `https://github.com/jsonch/starflow`

amount of hardware resources regardless of the number or form of measurement queries.

Three example monitoring applications demonstrate the flexibility and performance of `*Flow`: a host profiler that collects exact packet inter-arrival distributions; a traffic classifier that measures a large vector of complex flow statistics; and a microburst attributer that analyzes per-packet queue depths. These applications all measure complex and distinct metrics, and, with `*Flow`, they can all operate concurrently on the same stream of GPVs and dynamically change measurements without disrupting the network. Benchmarks show that the applications can scale to monitor Terabit-rate Internet workloads using under 10 cores of an Intel Xeon E5-2683 v4. This represents additional cost compared to `TurboFlow`, which can monitor similar workloads using a 4 core CPU, but is limited to simpler and fewer metrics.

To further demonstrate the flexibility of `*Flow`, we also introduce a simple adapter for executing Marple [11] traffic queries on GPV streams. The adapter, built on top of RaftLib [87], supports high-level query primitives (map, filter, groupby, and zip) designed for operator-driven performance monitoring. Using `*Flow` along with the adapter allows operators to run many different queries concurrently, without having to compile them all to the PFE or pause the network to change queries. Analysis shows that the `*Flow` PFE pipeline requires only as many computational resources in the PFE as *one* compiled Marple query. Currently, the adapter scales to measure 15-50 Gb/s of traffic per Xeon core, bottlenecked by overheads in our proof-of-concept implementation.

Contributions. This chapter has four main contributions. First, *grouped packet vectors*, a compact format for telemetry data that is efficient to process and gives a telemetry system high flexibility. Second, the design of a novel PFE cache data structure with dynamic memory allocation for efficient GPV generation. Third, the evaluation of a prototype of `*Flow` implemented on a readily available commodity P4 switch. Finally, four monitoring applications that demonstrate the practicality of

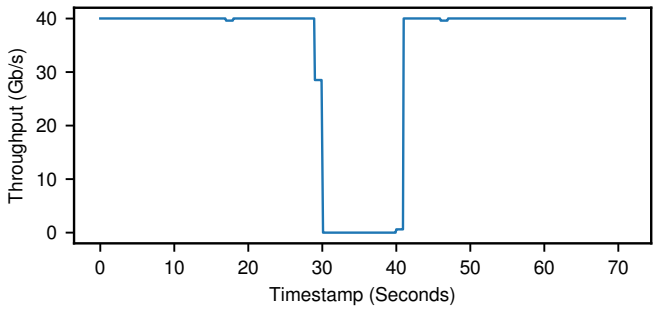

Figure 3.2: Network disruption from recompiling a PFE.

`*Flow`.

3.2 Flexibility Requirements and Challenges

For a telemetry system, flexibility requirements arise from the needs of the applications it supports. This section describes and motivates these requirements and explains why they are challenging to meet when a PFE is involved in metric calculation.

Concurrent Measurement. Concurrent measurement is the capability to calculate multiple metrics concurrently. It is important because telemetry systems support a diverse set of applications, which may need to analyze different metrics depending on their purpose.

Concurrency is a challenge because of the processing models and limited compute and memory resources available in a PFE. Each measurement query compiles to its own dedicated computational and memory resources in the PFE, to run in parallel at line rate. Computational resources are limited, which makes it challenging to fit more than a few metric queries at the same time. For example, each memory operation to state that persists beyond an individual packet requires a stateful ALU (sALU) [17], and a current generation PFE [19] has under 50.

Late Binding. Late binding is the capability to modify the set of metrics at runtime, i.e., while the network is operational, or even *after* telemetry data has been collected. Binding at runtime is important because measurement needs can change. For example, a network operator may add a new monitoring application to their network. Or, the user of an interactive debugging system [11] might pinpoint the root cause of a problem by analyzing metrics engineered to provide specific, problem-relevant information.

Being able to change metrics after runtime aids in the development and testing of new systems and models. The telemetry system performance models in Chapter 4 are an example of this. In their most basic form, the models predict the throughput of a hybrid telemetry system based on one packet feature – arrival timestamp. However, building these models required iteratively refining and testing complex mathematical and simulation-based code on previously collected datasets.

Late binding is an issue when metrics are compiled to a PFE. Changing metrics at runtime is difficult because in a reconfigurable ASIC, code is distributed across the compute elements in a pipeline. For example, each stage has its own instruction store and a programmable crossbar that determines which tables are applied to which packets. Changing this distributed code is challenging. Pausing the ASIC is one option, but that also halts forwarding for multiple seconds, as Figure 3.2 shows. Another option is provisioning the ASIC with multiple pipelines, so that it can hot swap from one pipeline to another. This increases chip space costs, i.e., that extra pipeline could instead be used to handle additional forwarding ports.

Changing metrics after runtime is impossible when metrics are compiled to a PFE, simply because transforming a batch of packet features into a set of flow metrics is a lossy, irreversible operation.

Complex metrics. Finally, many applications rely on metrics that are much more than just the computation of simple statistics. For example, flow classifiers use complex numerical operations [12], intrusion detection systems rely on neural networks

to calculate metrics [23], and debugging tools may need to correlate across flows at small timescales (Section 3.6).

A PFE cannot support complex metrics because of inherent compute and memory limitations that arise from the need to operate at high line rates with guaranteed latency while keeping costs (chip space and power) low.

- They cannot implement long sequential code because they process packets with a strict feed-forward pipeline of limited depth, e.g., around 16 stages [18, 17]. It is not possible to move backwards in the pipeline, so the maximum sequential code length is equal to the pipeline depth.

- Metrics that require more than a small, bounded amount of state cannot be supported because a sALU (which implements memory reads and writes) can only access one address per packet. Further, memory blocks are bound to specific sALUs at compile time and cannot be accessed by other sALUs.

- There are limited hardware primitives for numerical operations, e.g., floating point operations. Some operations could be implemented as a sequence of instructions, or approximated using match action tables [52]. But, these approaches are inefficient with respect to compute or memory limitations.

3.3 *Flow Overview

`*Flow` is a hybrid telemetry system that supports the flexibility goals of *concurrency*, *late binding*, and *complex metrics* and also the performance goals of high coverage and terabit-rate throughput using only a single PFE and backend server.

The main idea is to decouple metric calculation from packet feature selection and grouping. As Figure 3.3 shows, `*Flow` leaves packet feature selection and grouping on the PFE, but lifts metric calculation up to the telemetry system's backend server. This approach achieves high performance while significantly increasing flexibility.

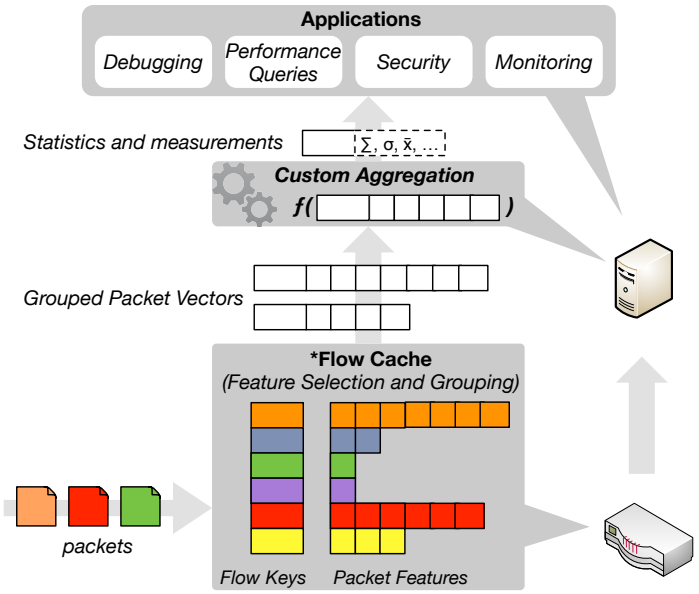

Figure 3.3: Overview of `*Flow`.

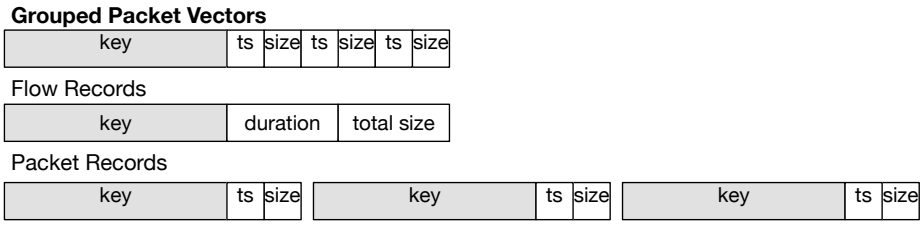

Figure 3.4: Comparison of grouped packet vectors, flow records, and packet records.

Performance. Performance is high because the bottlenecks that limit a server's throughput for selection and grouping ***do not affect calculation.*** Selection requires ingesting large packets at high rates, so its throughput is bound by the server's limited I/O bandwidth, as Chapter 2 demonstrated. Grouping requires looking up a flow for each packet, so its throughput is limited by a server's high memory latency. Calculation throughput, on the other hand, is limited only by the computational intensity of the metric. The metrics demonstrated by `TurboFlow` and prior PFE-based systems are all simple and, for such metrics, calculation is orders of magnitude faster than grouping. For example, a simple application that used the C++ `std::unordered_map` to group packet features by flow and then calculate the average packet length required 1535 cycles per packet for grouping, but only 45 cycles per packet for calculation. As

Chapter 4 shows, efficient implementations can reduce the per-packet calculation time to a few cycles and a server can leverage SIMD units to achieve high throughput even for surprisingly compute-intensive metrics.

Flexibility. All of the flexibility challenges described in Section 3.2 result from placing metric calculation in a PFE with inherent limitations due to its specialization. Moving calculation to a server increases flexibility because servers are designed for general workloads and thus do not share a PFE's limitations.

A server can support concurrent measurement of different metrics by time multiplexing its resources. Users can change metrics at run time by changing dynamically linked parts of the system, or after runtime by storing and replaying the digests from the PFE. Arbitrarily complex metrics can be calculated using a servers general compute capabilities.

System Design

Figure 3.3 shows the three main components of `*Flow`: grouped packet vectors, a `c`ache that generates them in a PFE, and a processing server that calculates metrics from the grouped packet vectors.

Grouped Packet Vectors. The key to decoupling feature selection and grouping from metric computation is the *grouped packet vector* (GPV) format for telemetry data exported from a PFE. A GPV stream is flexible because it contains per-packet features. Servers can measure different metrics from the same GPV stream and dynamically change measurement as needed, without impacting other measurement tasks or disrupting the PFE.

GPVs are also efficient to process. They are compact and small relative to the size of the underlying packet stream, which alleviates I/O bottlenecks. They also amortize the cost of flow lookups because each GPV summarizes multiple packets belonging to the same flow.

`*Flow` Cache. Switches with programmable forwarding engines [19, 17] (PFEs) compile the `*Flow` cache to their PFEs to generates GPVs. The cache is implemented as a sequence of match+action tables that applies to packets at line rate and in parallel with other data plane logic. The tables extract features tuples from packets; insert them into per-flow GPVs; and stream the GPVs to monitoring servers, using multicast if there are more than one.

GPV Processing. The `*Flow` cache streams GPVs to a `*Flow` agent running on a backend server. This can be a switch's integrated management server, a post-processing server that covers one or more switches in the network, or the server on which the monitoring applications themselves run. This chapter focuses on the last deployment model.

A `*Flow` agent running on a server receives GPVs from the switch, aggregates them into GPVs that describe flows over their full duration of activity, and delivers the complete GPVs into per-application queues. Each application defines its own metrics to compute over the packet tuples in GPVs and can dynamically change them as needed. As described above, metric calculation is efficient because GPVs alleviate the bottlenecks of I/O bandwidth and expensive flow lookups.

Further, if applications need packet-level visibility or the capability to change metrics later, the `*Flow` agent can forward them the raw GPV stream instead of computing metrics.

3.4 Grouped Packet Vectors (GPVs)

`*Flow` exports telemetry data from the switch in the grouped packet vector (GPV) format, illustrated in Figure 3.4, a new record format designed to support the decoupling of packet feature selection and grouping from metric computation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential packets in the respective flow. As Figure 3.4 shows, a GPV

is a hybrid between a packet record and a flow record. It inherits some of the best attributes of both formats and also has unique benefits that are critical for `*Flow`.

Similar to packet records, a stream of GPVs contains features from each individual packet and gives receivers the flexibility to calculate any metrics that can be derived from the underlying packet features. Typically, many metrics that are useful for a broad range of applications can derive from a small, common subset of packet features. For example, the 3 monitoring applications and 6 Marple queries we describe in Section 3.6 can all be derived from IP 5-tuples, packet lengths, arrival timestamps, queue depths, and TCP sequence numbers.

Although they provide the flexibility of packet records, GPVs are more efficient to process, similar to flow records. A GPV summarizes multiple packets from the same flow and deduplicates the IP 5-tuple. This reduces their size and, more importantly, amortizes the cost of flow lookups. A GPV stream provide these benefits, but unlike a flow record does not lock the receiver into specific metrics.

3.5 Generating GPVs

The core of `*Flow` is a telemetry cache that transforms a stream of packets into a stream of GPVs. It targets reconfigurable ASIC PFEs that process packets at guaranteed line rates exceeding 1 billion packets per second [17, 18, 88]. On these restricted platforms, the `*Flow` cache is more challenging to implement than `TurboFlow`'s metric cache because GPVs grow as they accumulate packets, which makes memory management more complex.

This section reviews the architecture of reconfigureable ASIC PFEs and describes how approximate eviction and dynamic memory allocation lets the `*Flow` cache fit into these restrictive processors.

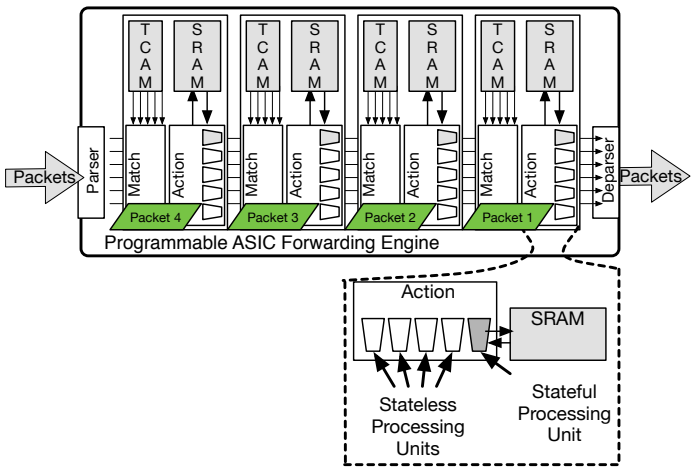

Figure 3.5: PFE architecture.

PFE Architecture

Figure 3.5 illustrates the general architecture of a PFE ASIC. It receives packets from multiple network interfaces, parses their headers, processes them with a pipeline of match tables and action units, and finally deparses the packets and sends them to an output buffer. PFEs are designed specifically to implement match-action forwarding applications, e.g., P4 [69] programs, at guaranteed line rates that are orders of magnitude higher than other programmable platforms, such as CPUs, network processors [89], or FPGAs, assuming the same chip space and power budgets. They meet this goal with highly specialized architectures that exploit pipelining, instruction level parallelism, and small parallel memories [17, 18].

It can be challenging to take advantage of PFEs for complex applications, especially those that require state persisting across packets, e.g., a cache. Persistent arrays, called "register arrays" in P4 programs, are stored in SRAM banks local to each pipeline stage. They are limited in three important ways. First, a program can only access a register array from the stage where it is stored. Second, each register array can only be accessed once per packet, using a *stateful ALU* that can implement simple programs for simultaneous reads and writes, conditional updates, and basic mathematical operations. Finally, all the sALUs in a single stage operate in parallel,

so there can be no sequential dependencies between stateful operations in the same stage. Work suggests that future PFEs can ease this restriction to support pairwise dependencies at the cost of increased chip space [17] or arbitrary dependencies at the cost of lower line rates [88], but these ideas have not yet been tested with hardware implementations.

Design

The `*Flow` cache performs two tasks. First, it selects features from each packet and transforms them into a simple but compact packet record format. Second, it accumulates packet records in per-flow GPVs. When the cache runs out of either *slots for new flows* or *buffers for the current GPV*, it evicts a GPV to the backend server for processing.

The high-level goal of the `*Flow` cache is to accumulate as many packet records into each GPV as possible, i.e., to minimize eviction rate. Lower eviction rates correspond with fewer memory operations in the backend telemetry server (and thus higher overall throughput) because each GPV only requires 1 flow lookup regardless of how many packets it summarizes.

Fitting the `*Flow` cache into a PFE requires limiting the amount of memory for flow slots and packet buffers. It also requires designing approximate cache eviction and memory allocation algorithms that are simple enough for the restrictive PFE to implement. Despite the limited memory and approximate algorithms, the `*Flow` cache achieves low enough eviction ratios to significantly reduce the workload of the backend telemetry server because of the inherent locality in underlying traffic. Chapter 4 explains the phenomena in more detail.

Cache (Re)Placement. A cache's placement policy determines where each object (each GPV, for `*Flow`) is placed, while its replacement policy decides which object should be evicted when a cache miss occurs. A PFE's restrictions on stateful opera-

tions limit the policies that it can support.

The simplest possible placement policy, which can be implemented in a PFE, is direct mapping. When a packet arrives, a hash of its flow key is taken. The hash is used as an index into an array of GPVs. If the GPV belongs to the same flow as the packet, it is updated to account for the new packet. If the GPV and packet belong to different flows, the GPV is evicted to the telemetry backend, and replaced with a new GPV for the packet.

The disadvantage of a direct map cache, compared with a fully associative cache that can place each object into any slot, is additional evictions due to collisions between hashes. Chapter 4 shows that scenarios where the number of flows is much larger than the number of cache slots, these collisions do not impact performance because it is dominated by misses due to insufficient cache capacity. For scenarios where capacity is sufficient, Chapter 4 introduces a skewed-associative cache [90] that significantly reduces collisions and can be implemented in a PFE.

Regardless of the placement policy, a small GPV cache can achieve a low eviction rate due to *temporal locality* in flows. Packets from a single flow tend to arrive in bursts (i.e., trains [31] or flowlets), where interarrival times for packets in the same burst are low, but interarrival times for packets in different bursts are high. This improves eviction rates by making it easier for multiple flows to share the same slot without frequently evicting each other.

Packet Buffer Allocation. In addition to managing GPV slots, the `*Flow` cache must also manage buffers that store the packet features in each GPV. Here, the PFE's restrictions on memory operations also limit possible algorithms.

The simplest possible approach is to statically allocate a fixed-width buffer for each cache slot. Whenever the buffer fills with packet feature vectors, its contents are evicted to the backend server as a GPV. This basic algorithm can be implemented for a PFE, and is evaluated in Section 3.7.

As Chapter 4 shows, static allocation requires significantly more memory to

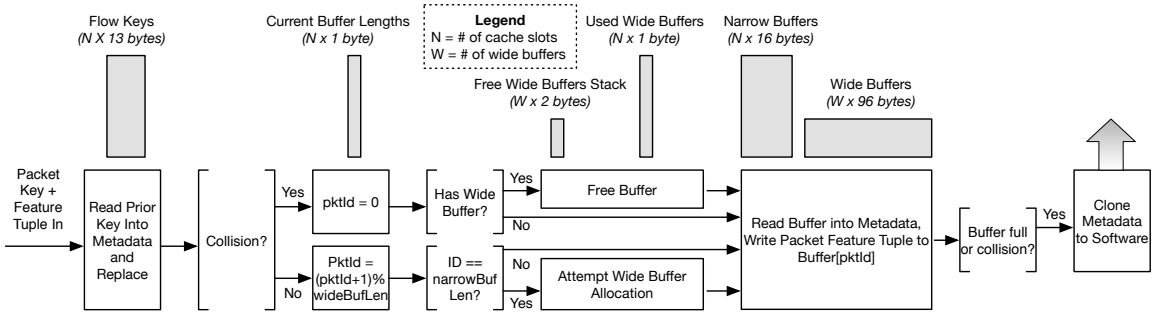

Figure 3.6: The `*Flow` cache as a match+action pipeline. White boxes represent sequences of actions, brackets represent conditions implemented as match rules, and gray boxes represent register arrays.

achieve the same eviction rates as dynamic allocation, i.e., allocating each packet buffer individually from a pool. Fully dynamic allocation is not possible in a PFE, but it can be approximated.

The core idea is to give each GPV slot the opportunity to allocate an additional fixed-width buffer when its packet count reaches a predefined threshold. When a GPV fills its initial fixed-width buffer for the first time (called the *narrow* buffer), the cache attempts to allocate the second buffer (called the *wide* buffer), drawn from a pool with fewer entries than there are cache slots. If the allocation succeeds, the slot utilizes both its narrow and wide packet feature buffers until its GPV is evicted. Otherwise, the slot utilizes only the narrow buffer until its GPV is evicted.

This approximation is effective because there is *source locality* in a multi-flow packet stream. In any given time interval, most of the packets arriving will be from a few highly active flows [35]. A flow that fills up its narrow buffer in the short period of time before it is evicted is more likely to be one of the highly active flows. Allocating a wide buffer to such a flow will reduce the average number of evictions due to packet buffer overfills, and thus the cache's overall eviction rate. Chapter 4 describes source locality and the effects of overfill evictions in more detail.

Implementation

Using the above heuristics for cache eviction and memory allocation, we implemented the `*Flow` cache as a pipeline of P4 match+action tables for the Tofino [19]. The implementation consists of approximately 2000 lines of P4 and Tofino-proprietary sALU code that implements the tables, 900 lines of Python code that implements a minimal control program to install rules into the tables at runtime, and a large library that is autogenerated by the Tofino's compiler toolchain. The source code is available at our repository[2] and has been tested on both the Tofino's cycle-accurate simulator and a Wedge 100BF-32X.

Figure 3.6 depicts the control flow of the pipeline. It extracts a tuple of features from each packet, maps the tuple to a GPV using a hash of the packet's key, and then either appends the tuple to a dynamically sized ring buffer (if the packet's flow is currently tracked), or evicts the GPV of a prior flow, frees memory, and replaces it with a new entry (if the packet's flow is not currently tracked).

We implemented the evict on collision heuristic using a simultaneous read / write operations when updating the register arrays that store flow keys. The update action writes the current packet's key to the array, using its hash value as an index, and reads the data at that position into metadata in the packet. If there was a collision, which the subsequent stage can determine by comparing the packet's key with the loaded key, the remaining tables will evict and reset the GPV. Otherwise, the remaining tables will append the packet's features to the GPV.

We implemented the memory allocation using a stack. When a cache slot fills its narrow buffer for the first time, the PFE checks a stack of pointers to free extension blocks. If the stack is not empty, the PFE pops the top pointer from the stack. It stores the pointer in a register array that tracks which, if any, extension block each flow owns. For subsequent packets, the PFE loads the pointer from the array before updating its buffers. When the flow is evicted, the PFE removes the pointer from

[2] `https://github.com/jsonch/starflow`

the array and pushes it back onto the free stack.

This design requires the cache to move pointers between the free stack and the allocated pointer array in both directions. We implemented it by placing the stack before the allocated pointer array, and resubmitting the packet to complete the free operation by pushing its pointer back onto the stack. The resubmission is necessary on the Tofino because sequentially dependent register arrays must be placed in different stages and there is no way to move "backwards" in the pipeline.

Configuration

Compile-time. The current implementation of the `*Flow` cache has three compile-time parameters: the number of cache slots; the number of entries in the dynamic memory pool; the width of the narrow and wide vectors; and the width of each packet feature tuple.

Feature tuple width depends on application requirements. For the other parameters, we implemented an OpenTuner [91] script that operates on a trace of packet arrival timestamps and a software model of the `*Flow` cache. The benchmarks in Section 3.7 show that performance under specific parameters is stable for long periods of time.

Run-time. The `*Flow` cache also allows operators to configure the following parameters at run-time by installing rules into P4 match+action tables. Immediately proceeding the `*Flow` cache, a filtering table lets operators install rules that determine which flows `*Flow` applies to, and which packet header and metadata fields go into packet feature tuples. After the `*Flow` cache, a table sets the destination of each exported GPV. The table can be configured to multicast GPVs to multiple servers and filter the GPV stream that each multicast group receives.

3.6 Processing GPVs

The *Flow cache streams GPVs to backend processing servers. There, a *Flow agent aggregates the GPVs from each flow into a single GPV that contains packet features for every packet in the flow. After reassembling a GPV, the *Flow agent delivers it to application-specific functions that can compute a wealth of potentially complex metrics. The functions can operate concurrently and calculate potentially complex metrics.

This section describes the *Flow agent, three motivating *Flow monitoring applications, and the *Flow adapter that supports interactive operator-driven network performance queries on GPV streams.

The *Flow Agent

The *Flow agent, implemented as a RaftLib [87] application, reads GPV packets from queues filled by NIC drivers and pushes them to application queues. While applications can process GPVs directly, the *Flow agent implements three performance and housekeeping functions that are generally useful.

Load Balancing. The *Flow agent supports load balancing in two directions. First, a single *Flow agent can load balance a GPV stream across multiple queues to support applications that require multiple per-core instances to support the rate of the GPV stream. Second, multiple *Flow agents can push GPVs to the same queue, to support applications that operate at higher rates than a single *Flow agent can support.

GPV Reassembly. GPVs from a *Flow cache typically group packets from short intervals, e.g., under 1 second on average, due to the limited amount of memory available for caching in PFEs. To reduce the workload of applications, the *Flow agent

can re-assemble the GPVs into a lower-rate stream of records that each represent a longer interval.

Cache Flushing. The `*Flow` agent can also flush the `*Flow` cache if timely updates are a priority, by sending a control packet down to the `*Flow` cache. The control packet specifies a single slot for the cache to evict. The `*Flow` agent can ensure that no GPV is cached for more than an approximate threshold period of time by tracking the last eviction time of each slot and sending control packets to the cache as needed.

*Flow Monitoring Applications

To demonstrate the practicality of `*Flow`, we implemented three monitoring applications that require concurrent measurement of traffic in multiple dimensions or packet-level visibility into flows. These requirements go beyond what prior PFE accelerated systems could support with compiled queries. With `*Flow`, however, they can operate concurrently and dynamically with low overhead.

The GPV format for the monitoring applications was a 192-bit fixed width header followed by a variable length vector of 32-bit packet feature tuples. The fixed width header includes IP 5-tuple (104 bits), ingress port ID (8 bits), packet count (16 bits), and start timestamp (64 bits). The packet feature tuples include a 20-bit timestamp delta (e.g., arrival time - GPV start time), an 11-bit packet size, and a 1-bit flag indicating a high queue length during packet forwarding.

Host Timing Profiler. The host timing profiler generates vectors that each contain the arrival times of all packets from a specific host within a time interval. Such timing profiles are used for protocol optimizers [83], simulators [92], and experiments [93].

Prior to `*Flow`, an application would build these vectors by processing per-packet records in software, performing an expensive hash table operation to determine which host transmitted each packet.

With `*Flow`, however, the application only performs one hash operation per GPV, and simply copies timestamps from the feature tuples of the GPV to the end of the respective host timing vector. The reduction in hash table operations reduces overhead.

Traffic Classifier. The traffic classifier uses machine learning models to predict which type of application generated a traffic flow. Many systems use flow classification, such as for QoS aware routing [84, 85], security [81, 5], or identifying applications using random port numbers or share ports. To maximize accuracy, these applications typically rely on feature vectors that contain dozens or even hundreds of different metrics [81]. The high cardinality is an obstacle to using PFEs for accelerating traffic classifiers, because it requires concurrent measurement in many dimensions.

`*Flow` is an ideal solution, since it allows an application to efficiently compute many features from the GPV stream generated by the `*Flow` cache. Our example classifier, based on prior work [94], measures the packet sizes of up to the first 8 packets, the means of packet sizes and inter-arrival times, and the standard deviations of packet size and inter-arrival times.

We implemented both training and classification applications, which use the same shared measurement and feature extraction code. The training application reads labeled "ground truth" GPVs from a binary file and builds a model using Dlib [79]; the classifier reads GPVs and predicts application classes using the model.

Micro-burst Diagnostics. This application detects micro-bursts [95, 96, 97], short lived congestion events in the network, and identifies the network hosts with packets in the congested queue at the point in time when the micro-burst occurred. This knowledge can help an operator or control application diagnose the root cause of periodic micro-bursts, e.g., `TCP` incasts [97], and also understand which hosts are affected by them.

Micro-bursts are difficult to debug because they occur at extremely small timescales,

e.g., on the order of 10 microseconds [98]. At these timescales, visibility into host behavior at the granularity of individual packets is essential. Prior to `*Flow`, the only way for a monitoring system to have such visibility was to process a record from each packet in software [99, 100, 101, 102] and pay the overhead of frequent hash table operations.

With `*Flow`, however, a monitoring system can diagnose micro-bursts by processing a GPV stream, making it possible to monitor much more of the network without requiring additional servers.

The `*Flow` micro-burst debugger keeps a cache of GPVs from the most recent flows. When each GPV first arrives, it checks if the high queue length flag is set in any packet tuple. If so, the debugger uses the cached GPVs to build a globally ordered list of packet tuples, based on arrival timestamp. It scans the list backwards from the packet tuple with the high queue length flag to identify packet tuples that arrived immediately before it. Finally, the debugger determines the `IP` source addresses from the GPVs corresponding with the tuples and outputs the set of unique addresses.

Interactive Measurement Framework

An important motivation for network measurement, besides monitoring applications, is operator-driven performance measurement. Marple [11] is a recent system that lets PFEs accelerate this task. It presents a high-level language for queries based around simple primitives (filter, map, group, and zip) and metric computation functions. These queries, which can express a rich variety of measurement objectives, compile directly to the PFE, where they operate at high rates.

As discussed in Section 3.2, compiled queries make it challenging to support concurrent or dynamic measurement. Using `*Flow`, a measurement framework can gain the throughput benefits of PFE acceleration *without* sacrificing concurrency or dynamic queries, by implementing measurement queries in software, over a stream of GPVs, instead of in hardware, over a stream of packets.

To demonstrate, we extended the RaftLib [87] C++ stream processing framework with kernels that implement each of Marple's query primitives on a GPV stream. A user can define any Marple query by connecting the primitive kernels together in a connected graph defined in a short configuration file, similar to a Click [103] configuration file, but written in C++. The configuration compiles to a program that operates on a stream of GPVs from the `*Flow` agent.

We re-wrote 6 example Marple queries from the original publication [11] as RaftLib configurations, listed in Table 3.3. The queries are functionally equivalent to the originals, but can all run concurrently and dynamically, without impacting each other or the network. These applications operate on GPVs with features used by the `*Flow` monitoring application, plus a 32 bit `TCP` sequence number in each packet feature tuple.

3.7 Evaluation

In this section, we evaluate our implementations of the `*Flow` cache, `*Flow` agent, and GPV processing applications. First, we analyze the PFE resource requirements and eviction rates of the `*Flow` cache to show that it is practical on real hardware. Next, we benchmark the `*Flow` agent and monitoring applications to quantify the scalability and flexibility benefits of GPVs. Finally, we compare the `*Flow` measurement query framework with Marple, to showcase `*Flow`'s support for concurrent and dynamic measurement.

All benchmarks were done with 8 unsampled traces from 10 Gbit/s core Internet routers taken in 2015 [104]. Each trace contained around 1.5 billion packets.

Chapter 4 generalizes these results to other workloads and measurement tasks and develops simple models to explain the underlying causes of `*Flow`'s high performance.

	Key Update	Memory Management	Pkt. Feature Update	**Total**
Computational				
Tables	3.8%	3.2%	17.9%	**25%**
sALUs	10.4%	6.3%	58.3%	**75%**
VLIWs	1.6%	1.1%	9.3%	**13%**
Stages	8.3%	12.5%	29.1%	**50%**
Memory				
SRAM	4.3%	1.0%	10.9%	**16.3%**
TCAM	1.1%	1.1%	10.3%	**12.5%**

Table 3.1: Resource requirements for `*Flow` on the Tofino, configured with 16384 cache slots, 16384 16-byte short buffers, and 4096 96-byte wide buffers.

The `*Flow` Cache

We analyzed the resource requirements of the `*Flow` cache to understand whether it is practical to deploy and how much it can reduce the workload of software.

PFE Resource Usage. We analyzed the resource requirements of the `*Flow` cache configured with a tuple size of 32-bits, to support the `*Flow` monitoring applications, and a maximum GPV buffer length of 28, the maximum length possible while still fitting entirely into an ingress or egress pipeline of the Tofino. We used the tuning script, described in Section 3.5, to choose the remaining parameters using a 60 second trace from the 12/2015 dataset [105] and a limit of 1 MB of PFE memory.

Table 3.1 shows the computational and memory resource requirements for the `*Flow` cache on the Tofino, broken down by function. Utilization was low for most resources, besides stateful ALUs and stages. The cache used stateful ALUs heavily because it striped flow keys and packet feature vectors across the Tofino's 32-bit register arrays, and each register array requires a separate sALU. It required 12 stages because many of the stateful operations were sequential: it had to access the key and packet count before attempting a memory allocation or free; and it had to perform the memory operation before updating the feature tuple buffer.

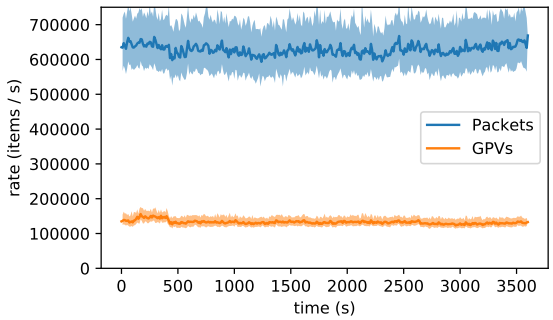

Figure 3.7: Min/mean/max packet and GPV rates in 2/2015 trace [104]. `*Flow` configuration: 16384 4-packet slots and 4096 24-packet extension slots.

Despite the high sALU and stage utilization, it is still practical to deploy the `*Flow` cache alongside other common data plane functions. Forwarding, access control, multicast, rate limiting, encapsulation, and many other common functions do not require stateful operations, and so do not need sALUs. Instead, they need tables and SRAM, for exact match+action tables; TCAM, for longest-prefix matching tables; and VLIWs, for modifying packet headers. These are precisely the resources that `*Flow` leaves free.

Further, the stage requirements of `*Flow` do not impact other applications. Tables for functions that are independent of `*Flow` can be placed in the same stages as the `*Flow` cache tables. The Tofino has high instruction parallelism and applies multiple tables in parallel, as long as there are enough computational and memory resources available to implement them.

PFE Resources Vs. Eviction Rate. Figure 3.7 shows the average packet and GPV rates for the Internet router traces, using the `*Flow` cache with the Tofino pipeline configuration described above. Shaded areas represent the range of values observed. An application operating on GPVs from the `*Flow` cache instead of packet headers needed to process under 18% as many records, on average, while still having access to the features of individual packets.[3] The cache tracked GPVs for an average

[3]18% is the ratio of the blue *Packets* line to the red *GPVs* line.

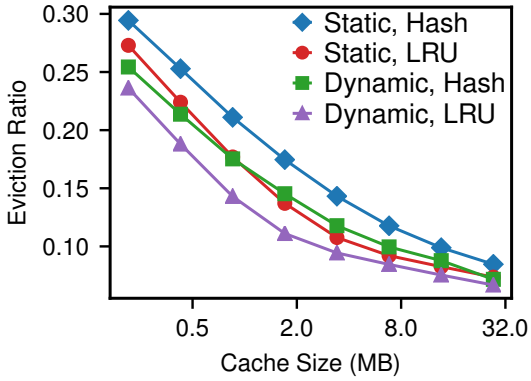

Figure 3.8: PFE memory vs eviction ratio.

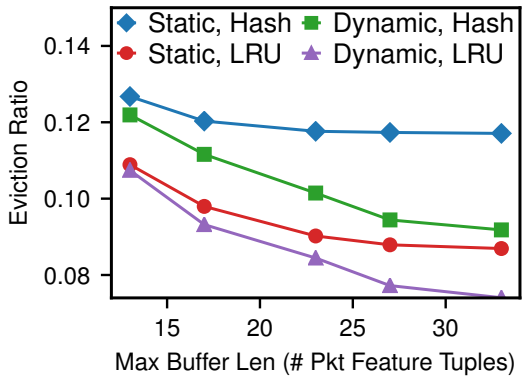

Figure 3.9: GPV buffer length vs eviction ratio.

of 640ms and a maximum of 131 seconds. 14% of GPVs were cached for longer than 1 second and 1.3% were cached for longer than 5 seconds.

To analyze workload reduction with other configurations, we measured *eviction ratio*: the ratio of evicted GPVs to packets. Eviction ratio depends on the configuration of the cache: the amount of memory it has available; the maximum possible buffer length; whether it uses the dynamic memory allocator; and its eviction policy. We measured eviction ratio as these parameters varied using a software model of the `*Flow` cache. The software model allowed us to evaluate how `*Flow` performs on not only today's PFEs, but also on future architectures. We analyzed configurations that use up to 32 MB of memory, pipelines long enough to store buffers for 32 packet feature tuples, and hardware support for an 8-way LRU eviction policy. Larger memories, longer pipelines, and more advanced eviction policies are all proposed features that are practical to include in next generation PFEs [18, 88, 11].

Figure 3.8 plots eviction ratio as cache memory size varies, for 4 configurations of caches: with or without dynamic memory allocation; and with either a hash on collision eviction policy or an 8-way LRU. Division of memory between and buffer slots between the narrow and wide buffers was selected by the OpenTuner script. With dynamic memory allocation, the eviction ratio was between 0.25 and 0.071.

This corresponds to an event rate reduction of between $4X$ and $14X$ for software, compared to processing packet headers directly.

On average, dynamic memory allocation reduced the amount of SRAM required to achieve a target eviction ratio by a factor of 2. It provided as much benefit as an 8-way LRU, but without requiring new hardware.

Figure 3.9 shows eviction rates as the maximum buffer length varied. Longer buffers required more pipeline stages, but significantly reduced eviction ratio when dynamic memory allocation was enabled.

Chapter 4.1 explains how locality in the underlying traffic causes these low eviction rates. It also generalizes the results, showing that for datacenters workloads the eviction ratio can be *an order of magnitude lower*.

`*Flow` Agent and Applications

We benchmarked the `*Flow` agent and monitoring applications, described in Section 3.6, to measure their throughput and quantify the flexibility of GPVs.

Experimental Setup. Our test server contained a Intel Xeon E5-2683 v4 CPU (16 cores) and 128 GB of RAM. We benchmarked maximum throughput by pre-populating buffers with GPVs generated by the `*Flow` cache. We configured the `*Flow` agent to read from these buffers and measured its throughput for reassembling the GPVs and writing them to a placeholder application queue. We then measured the throughput of each application individually, driven by a process that filled its input queue from a pre-populated buffer of reassembled GPVs. To benchmark multiple cores, we divided the GPVs across multiple buffers, one per core, that was each serviced by separate instances of the applications.

Throughput. Table 3.2 shows the average throughput of the `*Flow` agent and monitoring applications, in units of reassembled GPVs processed per second. For perspective, the average reassembled GPV rates for the 2015 10 Gbit/s Internet router

# Cores	Agent	Profiler	Classifier	Debugger
1	0.60M	1.51M	1.18M	0.16M
2	1.12M	3.02M	2.27M	0.29M
4	1.85M	5.12M	4.62M	0.55M
8	3.07M	8.64M	7.98M	1.06M
16	3.95M	10.06M	11.43M	1.37M

Table 3.2: Average throughput, in GPVs per second, for `*Flow` agent and applications.

traces, which are equal to their flow rates, are under 20 *thousand* per second [104]. The high throughput makes it practical for a single server to scale to terabit rate monitoring. A server using 10 cores, for example, can scale to cover over 100 such 10 Gb/s links by dedicating 8 cores to the `*Flow` agent and 2 cores to the profiler or classifier.

Throughput was highest for the profiler and classifier. Both applications scaled to over 10 M reassembled GPVs per second, each of which contained an average of 33 packet feature tuples. This corresponds to a processing rate of over 300 M packet tuples per second, around 750X the average packet rate of an individual 10 Gb/s Internet router link.

Throughput for the `*Flow` agent and debugging application was lower, bottlenecked by associative operations. The bottleneck in the `*Flow` agent was the C++ `std::unordered_map` that it used to map each GPV to a reassembled GPV. The reassembly was expensive, but allowed the profiler and classifier to operate without similar bottlenecks, contributing to their high throughput.

In the debugger, the bottleneck was the C++ `std::map` it used to globally order packet tuples. In our benchmarks, we intentionally stressed the debugger by setting the high queue length flag in every packet feature tuple, forcing it to apply the global ordering function frequently. In practice, throughput would be much higher because high queue lengths only occur when there are problems in the network.

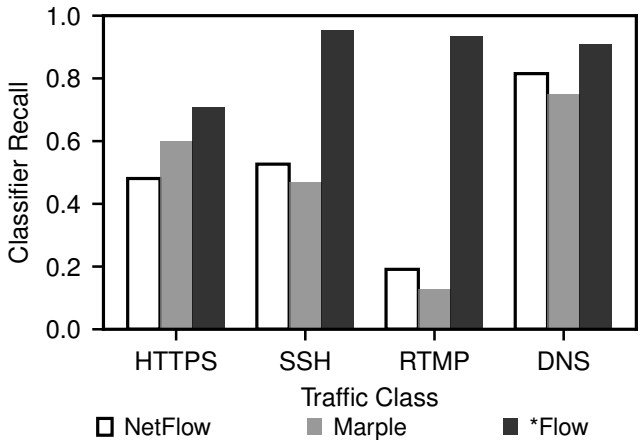

Figure 3.10: Recall of `*Flow` and baseline classifiers.

Classifier Accuracy. To quantify the flexibility benefits of GPVs, we compared the `*Flow` traffic classifier to traffic classifiers that only use features that prior, less flexible, telemetry systems can measure. The *NetFlow* classifier uses metrics available from a traditional NetFlow switch: duration, byte count, and packet count. The *Marple classifier* also includes the average and maximum packet sizes as features, representing a query that compiles to use approximately the same amount of PFE resources as the `*Flow` cache.

Figure 3.10 shows the recall of the traffic classifiers on the 12/2015 Internet router trace. The `*Flow` classifier performed best because it had access to additional features from the GPVs. This demonstrates the inherent benefit of `*Flow`, and flexible GPV records, for monitoring applications that rely on machine learning and data mining. Also, as Table 3.2 shows, the classifier was performant enough to classify >1 million GPVs per second per core, making it well suited to live processing.

Comparison with Marple

Finally, to showcase `*Flow`'s support for concurrent and dynamic measurement, we compare the resource requirements for measurement using compiled Marple queries against requirement using `*Flow` and the framework described in Section 3.6.

Configuration	# Stages	# Atoms	Max Width
`*Flow` cache	**11**	**33**	**5**
Marple Queries			
Concurrent Connections	4	10	3
EWMA Latencies	6	11	4
Flowlet Size Histogram	11	31	6
Packet Counts per Source	5	7	2
TCP Non-Monotonic	5	6	2
TCP Out of Sequence	7	14	4

Table 3.3: Banzai pipeline usage for the `*Flow` cache and compiled Marple queries.

PFE Resources. For comparison, we implemented the `*Flow` cache for the same platform that Marple queries compile to: Banzai [17], a configurable machine model of PFE ASICs. In Banzai, the computational resources of a PFE are abstracted as *atoms*, similar to sALUs, that are spread across a configurable number of stages. The pipeline has a fixed width, which defines the number of atoms in each stage.

Table 3.3 summarizes the resource usage for the Banzai implementation. The requirements for `*Flow` were similar to those of a *single* statically compiled Marple query. Implementing all 6 queries, which represent only a small fraction of the possible queries, would require 79 atoms, over 2X more than the `*Flow` cache. A GPV stream contains the information necessary to support *all* the queries concurrently, and software can dynamically change them as needed without interrupting the network.

Server Resources. The throughput of the `*Flow` analytics framework was between 40 to 45K GPVs/s per core. This corresponded to a per-core monitoring capacity of 15 - 50 Gb/s, depending on trace. Analysis suggested that the bottleneck in our current prototype is message-passing overheads in the underlying stream processing library that can be significantly optimized [106].

Even without optimization, server utilization with `*Flow` are similar to Marple, which required around one 8 core server per 640 Gb/s switch [11] to support measurement of flows that were evicted from the PFE early.

Chapter 4

Modeling Telemetry System Performance

A telemetry system transforms a stream of packets into a stream of flow metrics. High packet rates make performance critical and, as the systems introduced in this thesis demonstrate, a multi-level hybrid approach provides high performance without sacrificing flexibility.

This chapter develops a framework of models to better understand *why* hybridization is so effective for telemetry systems and evaluate the generality of the `TurboFlow`/ `*Flow` designs.

First, Section 4.1 focuses on the data plane cache in a hybrid telemetry system. It develops models that reveal the causal relationship between locality characteristics of packet streams and the benefit of a telemetry cache. These models enable a comparison of workloads, types of caches (i.e., `TurboFlow` versus `*Flow`), and eviction policies.

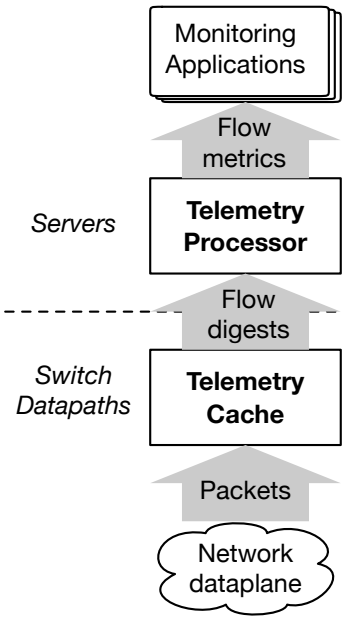

Figure 4.1: High level components of a hybrid telemetry system.

Second, Section 4.2 analyzes the server-based store/processor at the backend of a telemetry system. It models overall processing time by decomposing the backend's work into three high level tasks. Using these models, Section 4.2 explains the impact of telemetry cache configuration and metric complexity.

4.1 Modeling the Telemetry Cache

In a hybrid telemetry system, line-rate switch datapaths are used as caches that aggregate packets into per-flow digests for the telemetry processor. This amortizes the cost of expensive per-flow operations for the store, improving overall performance. The main goal of a telemetry cache is to summarize as many packets as possible into each digest, to maximize the amortization.

An Empirical Cache Model

The benefit of a telemetry cache depends on its configuration and characteristics of the underlying workload. Specifically, we focus on four factors:

- the number of concurrently active flows (F);
- the number of flows the cache can store (C);
- the packet inter-arrival time distribution of each flow f (I_f);
- and the eviction policy used by the cache.

To understand the relationship between these factors and cache performance, we begin by modeling a cache that emits fixed-width digests with per-flow statistics, i.e., `TurboFlow`. The cache's performance benefit in a finite-time interval can be quantified by its *packet to digest ratio* (D), i.e., the number of packets that arrive over the number of digests that it emits. The higher D, the more that the cache reduces the backing server's workload.

$$D = \frac{N_{pkt}}{N_{digest}} \tag{4.1}$$

$$N_{digest} = (1 - P) \cdot N_{pkt} \tag{4.2}$$

Assuming the cache operates at full utilization, i.e., a flow is tracked until the cache needs to make room for a new entry, N_{digest} depends primarily on the hit rate of the cache, P. Whenever a packet arrives from a flow that the telemetry system has observed but is not currently in the cache, the cache will evict a prior entry and send a digest describing it to the backing server.

We build an ***empirical model*** for P (and thus D) by extending the general cache models introduced by Garetto et al. [107]. They derive a set of closed-form equations to calculate P of an arbitrary cache by summing over the contribution from each object f that it may contain.

$$P = \sum_{f \in F} p_f * w_f \tag{4.3}$$

Here, w_f is flow's weight, or the probability of observing a packet from it. w_f can be calculated from I_f, the flow's packet inter-arrival time distribution: $w_f = 1/E[I_f]$.

p_f is the probability that flow f observes a hit. This depends on I_f and also Tc, the expected amount of time that a flow remains in the cache. For a random eviction policy, Garetto et al. [107] derive:

$$p_f = M_{I_f}(-1/Tc) \tag{4.4}$$

Here, M_{I_f} is the moment generating function of flow f's packet inter-arrival time distribution.

Making an I.I.D. assumption on the distribution Tc makes the system of equations tractable. Prior work has justified this assumption theoretically and empirically [107, 108, 109]. An I.I.D. distribution for Tc lets us calculate a *global* Tc that is independent of P by solving the following set of equations based on q_f, the probability that flow f is in the cache at a single point in time.

$$C = \sum_f E[\mathbb{1}(\text{f in cache})] = \sum_f q_f \tag{4.5}$$

$$q_f = w_f * Tc * (1 - Mr_{I_f}(-1/Tc)) \tag{4.6}$$

Equation 4.6 also assumes a random eviction policy. Equations for p_f and q_f with other common eviction policies are derived in [107]. Once Tc is calculated using the above equations, it can be plugged back into Equations 4.3 and 4.4 to solve for P, the hit rate of the entire cache.

Non-associative memory. The above equations model fully associative caches, where each item (or flow) competes for all cache slots. Due to hardware constraints,[1] datapath implementations of telemetry caches do not support associativity: each flow hashes to a single slot and competes for that slot against all the other flows with the same hash value.

We introduce a simple model extension to account for non-associativity. Observe that a non-associative cache for C flows is effectively an array of C single-flow caches, with flows statically load balanced based on hash.

With a fixed set of flows and a uniform hash function, the expected hit rate of a non-associative cache is the average over all possible assignments of flows to slots (A).

$$P = \frac{\sum_{a \in A} P_a}{|A|}$$

For an individual assignment a, the hit rate (P_a) is just the weighted average of per-cache hit rates, where the weight of cache i (w_i) is proportional to the fraction of packets it processed and its hit rate ($P_{(a,i)}$) can be calculated using the equations introduced above.

[1] Specifically, for a given packet, the datapath can only access 1 entry in each physical table. This constraint significantly reduces datapath cost in terms of chip space [17].

Trace	Capacity	Flow Rate	Packet Rate
Internet router link	10 Gb/s	5K - 40K	0.250 M - 0.6M
Datacenter leaf switch	1280 Gb/s	478K	124.0M

Table 4.1: Evaluation workloads.

$$P_a = \sum_{i \in [1,C]} w_i \cdot P_{(a,i)}$$

When there are many flows and cache slots, calculating the exact hit rate of a single-flow cache vector is intractable because there are a combinatorial number of possible flow to slot assignments. Instead, we approximate the expected hit rate by sampling assignments uniformly at random and calculating the average over a fixed number of trials.

Evaluation datasets. To validate the model, we compare the `TurboFlow` cache's packet to digest ratio with model estimates based on statistics measured in two datasets. Table 4.1 summarizes the datasets. First, 8 1-hour 10 Gb/s core router traces [47]. Second, 8 60-second traces from datacenter leaf switches in a synthetic network modeled using the YAPS [73] simulator.[2] We configured the simulator to model a two-tier network with 12 (8 leaf, 4 spine) 1.4 Tb/s switches and 144 hosts that generate flows with packet inter-arrival times matching a Facebook web cluster [33].

We divide each trace into trials of duration t. In each trial, we measure the D value of `TurboFlow` and three model parameters: C, the number of flow slots; F, the number of active flows in the interval; and I_f, the empirical packet inter-arrival distribution of each flow.

[2] YAPS is a simple event driven simulator that models packet arrival distributions, switch processing, queueing, and transmission delays. It has also been used in other recent work [75] with similar arrival distribution configurations.

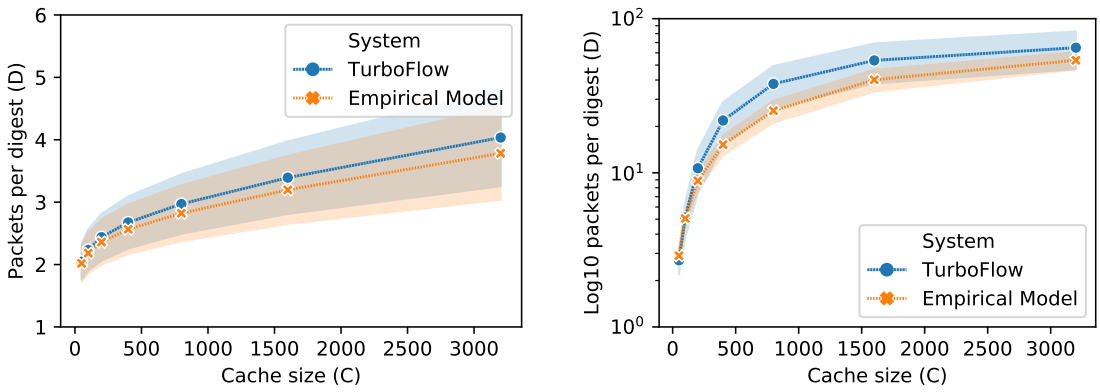

Figure 4.3: Observed and modeled performance of the `TurboFlow` cache in Internet (left) and datacenter (right) traces. Markers show averages and error bands show standard deviation.

The selection of t, trial duration, balances two concerns. If we use too short of a duration, there will not be enough samples for accurate predictions. If we use too long of a prediction, model accuracy will drop because it assumes that all flows are active over the entire interval, which is not true over long timescales.

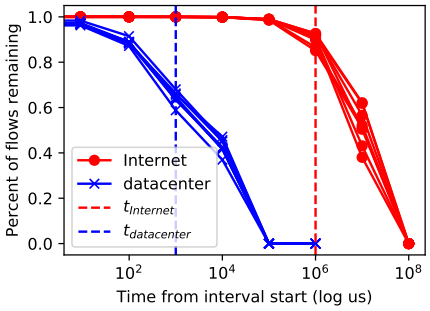

Figure 4.2: Flow churn is used to select t, the duration of trials in our experiments.

We choose t by identifying the largest possible value that meets a simple constraint: most flows active at the beginning of an interval are still active at the end of the interval. A coarse exponential parameter sweep from $t = 10^1$ us to $t = 10^8$ us finds that $t = 10^3$ us is the largest value for which the constraint holds in the datacenter traces, while $t = 10^6$ us is the largest value for the Internet trace, as Figure 4.2 shows.

Validation with `TurboFlow`. Figure 4.3 plots D as cache size increases in these trials. Quantitatively, the model is accurate and conservative. For the Internet traces, over 90% of model predictions are within the range of observed values for the same

cache size, model variance is within 7% of system variance, and the average predicted packets per digest is off by a maximum of 6.5%, for a cache size of 3200. In the datacenter traces, the model underestimates D to a greater extent because a larger percentage of flows stop before the end of the trial (Figure 4.2).

Most importantly, the model correctly predicts three important performance trends.

1. A small cache provides a large workload reduction (e.g., a 100 flow cache has a digest to packet ratio of approximately 0.4);

2. For the Internet workload, benefit increases logarithmically with cache size.

3. For the datacenter workload, benefit is higher and increases linearly with cache size.

In the next sections, we use the model to explore these results and better understand *why* a telemetry cache is so beneficial in both Internet and datacenter scenarios.

A locality-parameterized Cache Model

An important (and somewhat surprising) observation, in both the model and the implementation, is that a small telemetry cache has a high performance benefit. This is most evident in the Internet workloads. In any 1 second interval of these traces, there are between 10000-50000 concurrently active flows. A `TurboFlow` cache with only 100 slots (room for only 0.3% of all concurrent flows) observes a hit rate of around 60% (C=100 in Figure 4.3). Typically, a cache needs to hold much larger percentage of objects to achieve this high of a hit rate. For example, a youtube request cache needs to store approximately 10% [107] of all objects to provide a 50% hit rate.

Why does a small cache provide such a disproportionately high benefit in a telemetry system?

Based on prior analysis in other domains, we expect that it is related to locality in the underlying workload [110, 107]. At a high level, locality means that if the cache

has recently observed a packet from a flow, it is much more likely to observe another packet from that flow again in the near future. In this section, we define and quantify axes of locality in a packet stream, extend the empirical model into a ***parametric model*** that predicted hit rate based on locality parameters, and use the model to characterize the relationship between locality and hit rate.

Characterizing Locality. In a telemetry cache's workload, there are two types of locality that benefit a telemetry cache: *source locality* and *burst locality*. We describe each axis of locality and show how it can be quantified.

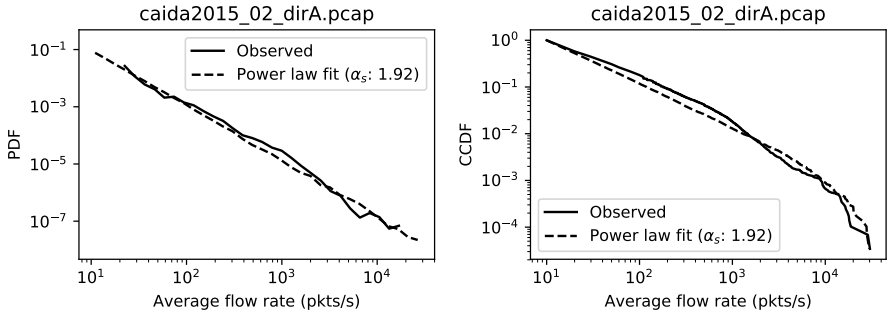

Figure 4.4: PDF of flow rates in a 10 Gb/s Internet router trace (log-log scale). The data has a heavy tail and fits a power-law distribution ($P[X] \approx \frac{1}{(X^{\alpha})}$).

First, *source locality*: average link utilization is dominated by a small fraction of high bandwidth ("heavy hitter") flows. This benefits the cache because inter-arrival times are lower in heavy hitters, which makes them more likely to observe cache hits.

We can characterize source locality based on the distribution of average flow rates. Figure 4.4 shows an example – the PDF and CCDF of flow rates observed in one second of the 10 Gb/s Internet router trace. Most flows ($> 90\%$) had rates lower than 100 packets/s. The heavy hitter flows in the remaining 10%, however, had rates of up to 50,000 packets/s.

As Figure 4.4 shows, the flow rate data fits a power-law distribution[3] with 1 parameter ($P[X] \approx 1/X^{\alpha_s}$). The decay coefficient, α_s, quantifies the degree of source

[3]Distribution fit found using the `powerlaw` library.

locality. Lower values correspond to a greater degree of source locality, i.e., heavy hitters of higher magnitude are more likely.

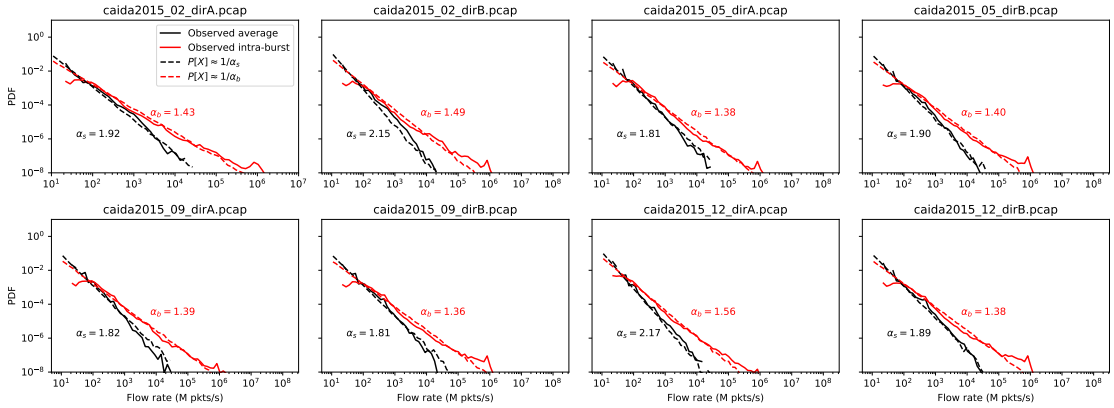

Figure 4.5: Best-fit curves for average and intra-burst flow rates in the first second of each 10 Gb/s Internet router trace.

In addition to source locality, a multi-flow packet stream also exhibits *burst locality*: packets in a flow, especially a heavy hitter, arrive in bursts or trains [31]. During a burst, a flow's rate is much higher than average. A telemetry cache can exploit this to use memory more efficiently – when a burst ends for one flow, the cache can increase its hit rate by evicting the flow and using the memory to track other flows.

We can characterize burst locality in a multi-flow packet stream based on the distribution of *intra-burst rates*, i.e., average flow rates *during bursts*. For a flow f, we define a burst as a sequence of consecutive packets with inter-arrival times in the bottom nth percentile for that flow. We measure per-flow intra-burst rates and fit them to a power-law distribution ($P[X] \approx 1/X^{\alpha_b}$). Here, the decay coefficient (α_b) quantifies the degree of burst locality; lower values correspond to the presence of flows with higher burst rates and thus more locality.

Figure 4.5 shows the intra-burst rate distributions of 8 10 Gb/s Internet router traces,[4] with average flow rates shown for comparison. The decay of the intra-burst rate distribution, parameterized by α_s, is significantly lower than the decay of the av-

[4]Here, we define a burst as an sequence of packets with inter-arrival times in the top 90th percentile.

erage rate distribution. At the tail, intra-burst rates are nearly 2 orders of magnitude higher than average rates.

A locality-based cache model. Together, α_s and α_b quantify two important and orthogonal axes of locality in a multi-flow packet stream. Using these statistics, we can explore the relationship between locality and cache performance by extending our cache model into a ***parametric model*** that predicts hit rate based on α_s and α_b, rather than a vector of per-flow distributions. The parametric model has only 4 parameters: number of flows (F), number of cache slots (C), source locality statistic (α_s), and burst locality statistic (α_b).

The model extends the empirical model in Section 4.1. The core idea is to use α_s and α_b to generate per-flow parameters that the empirical model can take as input. There are two such parameters: w_f, the weight of flow f; and I_f, the packet inter-arrival time distribution of flow f.

w_f is the probability of observing a packet from f during the entire modeled operational period of the cache. By definition, it is proportional to the average rate of f, which makes it straightforward to generate w_f from α_s. First, sample a vector r of F average rates from the power law distribution $P[X] \approx 1/X^{\alpha_s}$. Second, with r, calculate the weight of flow f as:

$$w_f = \frac{r_f}{\sum r} \tag{4.7}$$

I_f, the packet inter-arrival time distribution of flow f, is used in Equations 4.4 and 4.6 to calculate the probability that a burst of packets from flow f have inter-arrival times lower than Tc, the cache's expected eviction time. The parametric model approximates I_f using α_b. It samples a vector b of F burst rates from the power law distribution $P[X] \approx 1/X^{\alpha_b}$ and uses b_f to model I_f as an exponential distribution:

$$\hat{I_f}\ Exp(\lambda = b_f) \tag{4.8}$$

We choose the exponential distribution because of its simplicity and prior work that shows it can accurately model packet inter-arrival times over small timescales (i.e., the timescales of bursts) [111].

We can bound the approximation ratio of the terms that depend on I_f. Equations 4.4 and 4.6 use I_f's moment generating function (MGF), $M_{I_f}(-1/T)$, where T is the expected cache eviction time. To bound the approximation ratio, we define a burst as a sequence of packets in a flow with inter-arrival times that do not exceed the p-th percentile inter-arrival time of the flow.

With this definition of a burst, we can bound the approximation ratio to:

$$\frac{M_{I_f}(-1/T)}{M_{\hat{I_f}}(-1/T)} \leq 1/p \tag{4.9}$$

Proof: A MGF returns a 0 - 1 value. For an empirical distribution of inter-arrival times X defined by n samples X_i $(0 \leq i \leq n)$, the MGF is:

$$M_X(t) = \frac{1}{n} \sum e^{X_i t}$$

We can rewrite this with terms corresponding to intra and inter-burst components, i.e., below and above the percentile threshold p:

$$M_X(t) = p \sum_{i<p} e^{X_i t} + (1-p) \sum_{j>=p} e^{X_j t}$$

Assuming that the exponential distribution of inter-arrival times, $\hat{X}$, accurately models the empirical distribution defined by the samples X_i, we can expand the approximation ratio to:

$$\frac{M_X(t)}{M_{\hat{X}}(t)} = \frac{p \sum_{i<p} e^{X_i t} + (1-p) \sum_{j>=p} e^{X_j t}}{p \sum_{i<p} e^{X_i t}}$$

In the model, the input t to the MFG is $-1/T$, where T is the expected cache eviction time. The approximation ratio becomes:

$$\frac{M_X(-1/T)}{M_{\hat{X}}(-1/T)} = \frac{p \sum_{i<p} e^{-X_i/T} + (1-p) \sum_{j>=p} e^{-X_j/T}}{p \sum_{i<p} e^{-X_i/T}}$$

In this equation, $\sum_{i<p} e^{-X_i/T} > \sum_{j>=p} e^{-X_j/T}$ because $X_i < X_j$. Thus:

$$\frac{M(-1/T)}{M_{\hat{X}}(-1/T)} = \frac{p\sum_{i<p} e^{-X_i/T} + (1-p)\sum_{j>=p} e^{-X_j/T}}{p\sum_{i<p} e^{-X_i/T}} \leq$$
$$\frac{p\sum_{i<p} e^{-X_i/T} + (1-p)\sum_{i<p} e^{-X_i/T}}{p\sum_{i<p} e^{-X_i/T}} =$$
$$\frac{p+(1-p)}{p} = \frac{1}{p}$$

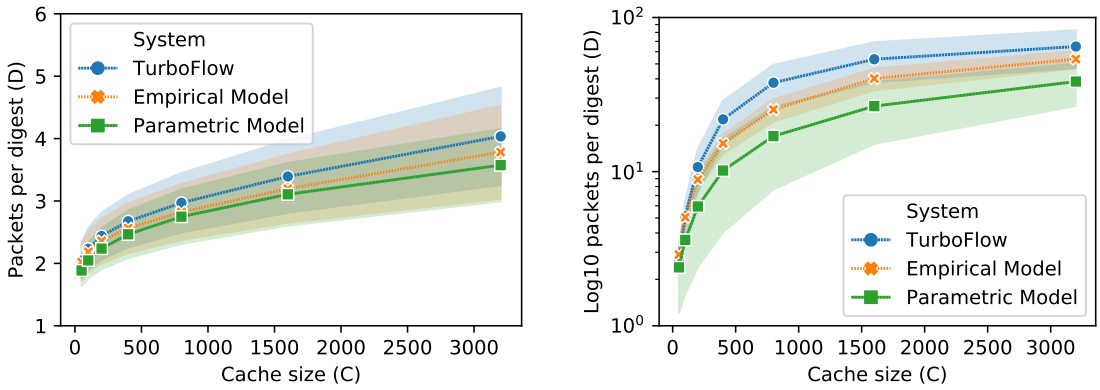

Figure 4.6: Observed and parametric modeled performance of the `TurboFlow` cache in Internet (left) and datacenter (right) traces. Markers show averages and error bands show standard deviation.

Validation. Figure 4.6 shows the hit rates predicted by the parametric model. The simple, 5 parameter locality model captures the overall performance trend. However, it sacrifices accuracy, particularly in the datacenter traces where the parametric model is off by up to 50%. This is the cost of modeling the complex inter-arrival distributions of hundreds of concurrent flows using only two parameters. Importantly, the model is conservative and underestimates real system performance. It only predicted values larger than observed by the real system in 2% of the trials. This lets us have high confidence that, in a workload with given α_s and α_b statistics, a telemetry cache will perform *at least* as well as the model predicts.

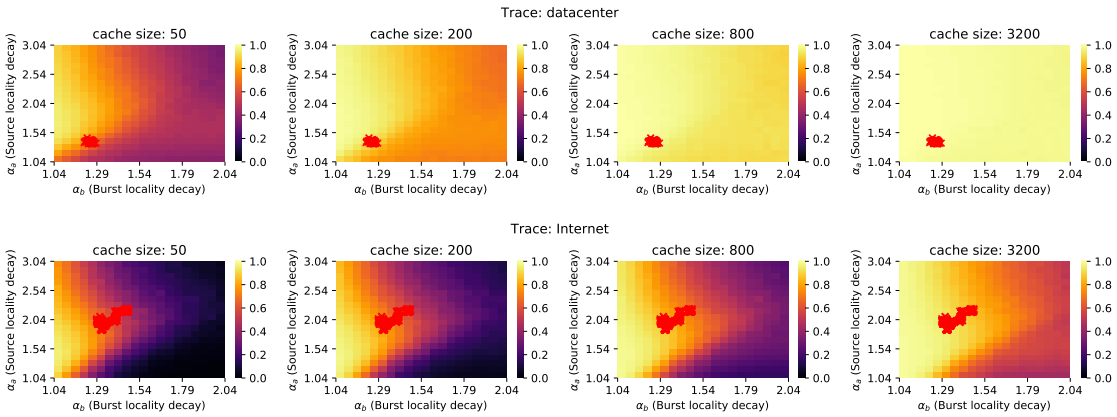

Figure 4.7: Heat maps of modeled telemetry cache hit rate as source and burst locality parameters (α_s and α_b) vary. Higher values mean less locality. Red X's indicate best fit parameters of trials in the evaluation datasets, with datacenter trials in the top row and Internet trials in the bottom row.

Analysis. Using the model, we can sweep the range of possible α_s and α_b parameters to show how cache hit rate changes with locality. Figure 4.7 shows results for the datacenter and Internet traces, with whiter color representing better (higher) hit rates. Source and burst locality are both important, for both data sets and especially when cache sizes are small.

More burst locality (lower α_b values on the Y axis) always improves performance. However, there is an optimal region for source locality (α_s). This is because as α_s decreases, the number and magnitude of heavy hitters increases. Cache performance improves until the number of heavy heavy hitters exceeds the size of the cache, which causes an inflection point. The inflection point reduces with α_b (i.e., stronger locality) because additional burst locality allows the cache to share memory more efficiently and tolerate more heavy hitters. With larger caches, the significance of the inflection point reduces.

Hit rate is high for a wide range of locality parameter values in both the Internet and datacenter traces. For the Internet traces (represented by the red X's in the bottom row of Figure 4.7), the contributions of source and burst locality are smooth and additive. For the cache to not have any benefit (e.g., a hit rate $< 10\%$), burst

locality would have to be much lower and source locality would have to be either significantly lower or higher.

The datacenter traces (represented red X's in the top row of Figure 4.7) have more source locality because they are dominated by large object transfers. This puts them closer to the inflection line. However, even beyond the inflection line, hit rate remains high in the datacenter.

Summary. In summary, the parametric model allows us to explain telemetry cache performance in terms of flow count, source locality, and burst locality. Analysis with the model shows that all parameters are important, but that the benefit of a telemetry cache does not depend on any one parameter. Analysis with the model shows that telemetry cache performance is high for a wide range of parameter values, demonstrating the generality of their benefit.

Improving Memory Efficiency in Datacenters

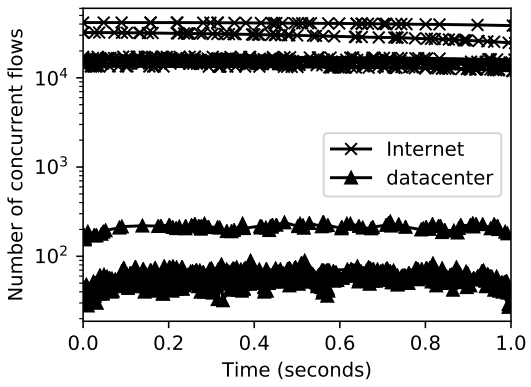

Figure 4.8: Number of concurrently active flows in the Internet (top) and datacenter (bottom) traces.

As Figure 4.3 shows, telemetry cache performance is much higher in datacenter traces versus Internet traces. This effect is due to the lower number of concurrently active flows in the datacenter. Figure 4.8 compares active flows in the datacenter leaf

switches with individual 10 Gb/s Internet links. A leaf switch observes orders of magnitude fewer flows because it is nearer to the edge of the network and connects fewer endpoints than the core Internet link.

In such environments, the telemetry cache can be sized large enough to track *all* concurrently active flows and reduce the telemetry backend's workload to processing only O(1) digests per flow. For example, we observe an average of only 250 concurrently active flows in our datacenter leaf switch traces, a fraction of what a telemetry cache can fit.

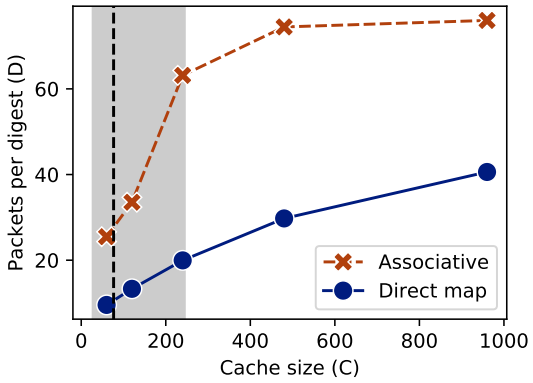

Figure 4.9: Comparison of a direct map cache and an associative cache with random replacement in a datacenter workload, using the empirical model (Section 4.1). Markers show averages across 45 trials from 9 traces. The vertical dashed line marks the average number of concurrent flows across all trials and the shaded area shows the range of per-trial averages.

When a cache's capacity can be large relative to its working set, associativity can significantly improve performance by eliminating evictions due to collisions. We can test the expected effect using the empirical model from Section 4.1. Figure 4.9 compares modeled performance of a non-associative cache (like the datapath implementations of `TurboFlow` and `*Flow`) against that of an associative cache with a random replacement policy.

When cache size is approximately equal to the number of concurrently active flows flows (i.e., the shaded area of Figure 4.9), the associative cache model has a 3X higher

packets per digest ratio than the direct map cache model. Further, the direct map cache needs an average of 5X more slots (C) to achieve the same hit rate as the associative cache.

Direct mappped caches have worse performance because of evictions due to collision. Each flow can only be placed in a single slot based on its hash value. When two or more concurrent flows have the same hash, they will map to the same slot and repeatedly evict each other. In an associative cache, there are no evictions due to collisions. Each flow can map to *any* slot and evictions only occur when the cache runs out of capacity.

Reducing Collisions By Layering. Guided by the model and prior results on combinatorial optimization [112], we introduce a mapping scheme that significantly reduces collisions in restrictive datapath implementations.

The core idea is to use multiple direct map caches to implement a *multi-hash cache*. Instead of using the datapath's memory as a single direct map cache, split it into a pipeline of N smaller direct map caches, or "*layers*", each of which uses an independent hash function $H_1, ...H_N$. When a collision occurs in layers 1 through $N-1$, the victim flow is inserted into the subsequent layer, where it may evict another flow. In the final layer N, a collision causes the final victim to be evicted to backing store, just as it would have in a direct map.

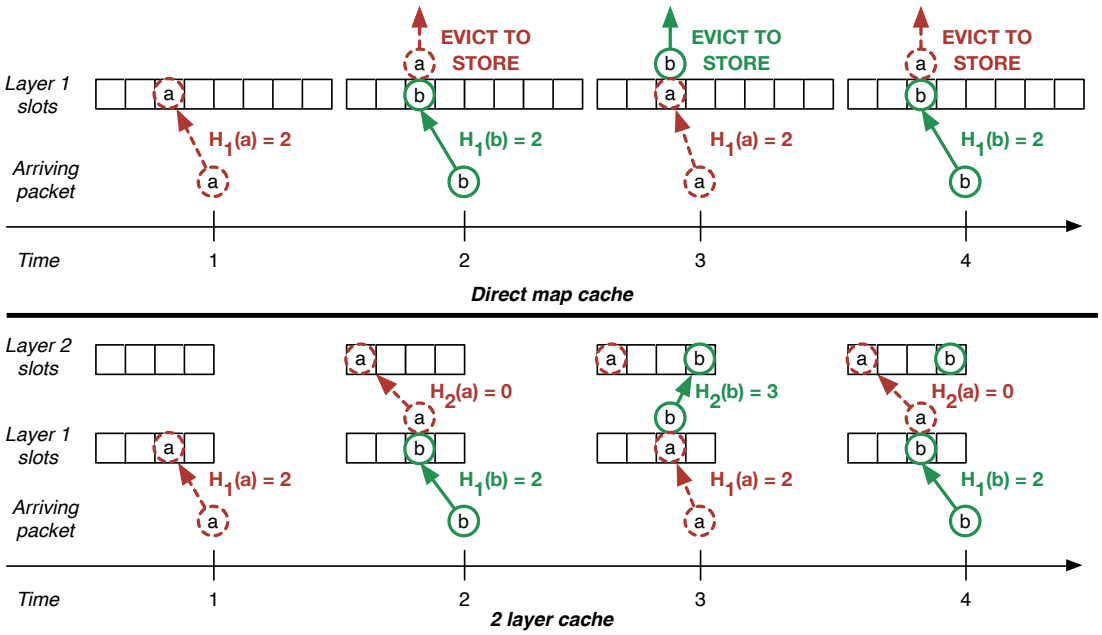

Figure 4.10: Example of cache behavior with 2 colliding interleaved flows using direct mapping (top) and 2-layer mapping (bottom). For these example flows, the second layer resolves the collision internally.

As prior work in other domains has shown [90, 113], a multi-hash cache reduces collisions by providing inter-layer dispersion: even if two objects collide in one layer, they are unlikely to collide in the next. Consider the simple example in Figure 4.10. It depicts a cache with 8 slots and 2 flows, a and b, with interleaved packets. Organized as a direct map, the probability of the flows colliding, i.e., mapping to the same slot, is:

$$P[(H_1(a) == H_1(b))] = \frac{1}{8} \tag{4.10}$$

As the top panel illustrates, if the two flows collide, the direct map cache has no way to resolve. Whenever a packet from either flow arrives, an eviction to the backing store occurs.

Organizing the slots into two layers, illustrated by the bottom panel of Figure 4.10, gives the cache a mechanism to resolve collisions internally. If a collision occurs in

the first layer, the second layer has a high chance of resolving it because it is likely that a and b map to different second layer slots, i.e., $P[H_2(a)! = H_2(b)]$ is high. In this case, there are no evictions from the cache to the backing store, only from the first cache layer to the second.

For a pair of flows to collide in a way that a 2 layer cache cannot resolve, they must collide for both H_1 *and* H_2. Assuming that these are independent hash functions, the probability is:

$$P[(H_1(a) == H_1(b)) \cap (H_2(a) == H_2(b))] =$$
$$P[(H_1(a) == H_1(b))] \cdot P[(H_2(a) == H_2(b))] = \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16} \tag{4.11}$$

Comparing Equations 4.11 and 4.10, we see that organizing the 8 slot cache into two 4 slot layers halves the probability of two arbitrary flows colliding. Generalizing these Equations, the probability of two flows colliding in a way that a cache with C slots divided into L layers cannot resolve is:

$$\prod_{l=1}^{L} P[(H_l(a) == H_l(b))] = \prod_{l=1}^{L} \frac{1}{(C/L)} = \frac{1}{(C/L)^L} = \left(\frac{L}{C}\right)^L \tag{4.12}$$

This probability decreases exponentially with the number of layers and larger caches observe more benefit from organization into layers. This simple analysis supports our intuition that a multi-layer cache should reduce the frequency of eviction due to collisions. Prior work describes the principles underlying multi-hash caches in more detail [90, 113, 112]; we leave a more complete analysis of our layered variant for future work.

Evaluation. The benchmarks plotted in Figure 4.11 evaluate a software implementation of `TurboFlow` with multi-layer caches running on the datacenter traces. As expected, the benefit of partitioning the direct map cache into multiple layers is significant. Going from a direct map to a 2-layer cache improves D by around $3X$ on average, while a 4-layer cache improves D by up to $6X$ (for $C = 240$).

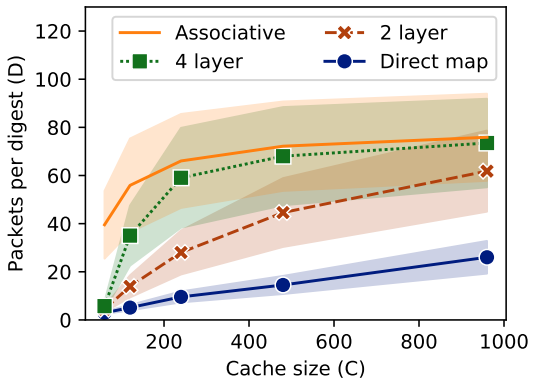

Figure 4.11: Packets per digest generated by `TurboFlow` with a multi-layered cache in datacenter workloads. All configurations use the same amount of total memory. Lines show averages and band show standard deviation. The associative cache uses a random replacement policy.

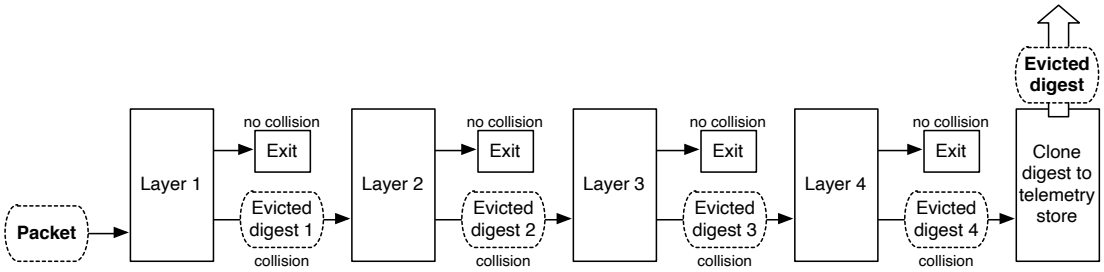

Figure 4.13: A multi-layer cache can be implemented as a pipeline of simple direct map caches.

In the benchmarks, a 4-layer cache performs 60% as well a fully associative cache when $C = 120$. The D values of 4-layer and fully associative caches quickly converge above that point. These results suggest that multiple layers are an effective replacement for associativity whenever a cache can be sized large enough to store approximately 2X the number of concurrently active flows (74 for these traces).

On the other hand, when C is smaller than the number of concurrent flows, there is a significant gap between the 4-layer and fully associative caches. For example, at C=60 there is a 6.8X gap between the two.

Implementation. To implement a multi-layer cache in a line rate datapath, we simply pipeline multiple, direct-map caches. Figure 4.12 shows the logic of a single layer and Figure 4.13 shows the design of a 4 layer cache. An individual layer is simply a direct map telemetry cache, like `TurboFlow` or `*Flow`, that accepts an packet or digest as input and generates a digest as output that is emitted if there is a collision. Each layers output digest is processed as input by the subsequent stage and the last layer's output is sent to the telemetry backend server.

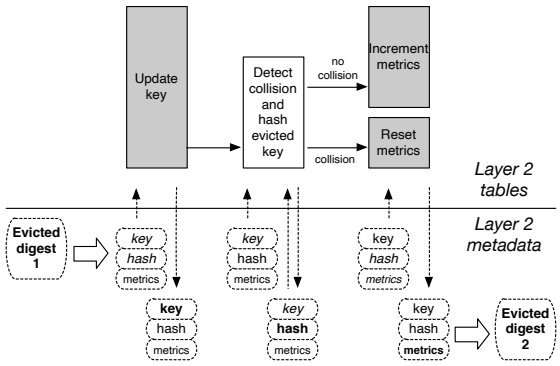

Figure 4.12: One layer of a direct map cache. (Layer 2 in Figure 4.13).

For the Barefoot Tofino with 12 ingress stages, we can exactly fit a 4-layer cache (with the final export stage placed in egress). The cache uses 4 stages for key lookups, 4 stages for collision detection, and 4 stages for value updates. Maximally sized, it can support a total of 176K slots with 128b keys and values. Slot count is limited by per-stage SRAM, while key and value widths are limited because each stage can only perform 4 32-bit memory operations.

The most significant cost of adding multiple layers is less flexibility to trade cache "height" (number of flow slots) for cache "width" (size of per-flow state). For example, the 4-layer configuration described above can only support up to 128b of per flow state, regardless of how short the cache is. A 2-layer configuration, however, could use the same amount of memory to support 176K slots with 128b keys and 128b values or 88K slots with 128b keys and 384b values. We leave a detailed study of flexibility trade-offs and other multi-layer optimizations for future work.

Finally, note that the additional stage requirements do not impact data plane programs that are independent of telemetry (e.g., forwarding and load balancing). These programs are independent of a telemetry cache so can run in parallel on the datapath's other hardware resources.

Header Caches

The sections above model a *metric cache* that exports digests containing summary statistics. In this section, we compare metric caches against *header caches* (e.g., `*Flow`) that export digests containing a subset of each packet's header fields. This lets the calculation be done entirely by the backend processor of the telemetry system for increased flexibility.

However, the extra flexibility has performance costs. If the expected number of packets per digest is high, more memory may be required to store a vector of per-packet records instead of flow metrics. This overhead is higher in switch datapaths that cannot support dynamic memory allocation. In this section, we model and analyze the overheads of header caches with dynamic and static memory allocation policies.

Dynamic Memory Allocation. A header cache can achieve the same D ratio as a metric cache, but may require additional memory. The overhead is lowest if the header cache can dynamically allocate packet header buffers to flow slots at run time. This is possible for implementations on middlebox servers or NPU line card [44]. However, dynamic allocation at line rate is not possible on the current generation of reconfigurable ASICs [17].

To derive a simple, conservative model of the expected memory utilization for a dynamic header cache, we observe that the number of packet record buffers allocated to a slot is highest immediately before an eviction. At this point, the slot has D buffers, one for each packet header in the digest it is about to emit. Thus, a cache slot's expected memory utilization for packet records is at most $D * W_{pkt}$, where W_{pkt} is the size of a single packet record.

Every cache slot also stores flow features (e.g., flow key, first timestamp, packet counters) in a fixed width field of size W_{flow}. Summing the fixed and per-packet terms and multiplying by C, the number of slots in the cache, we get a simple equation

that models the maximum expected memory utilization of a dynamically allocating header cache.

$$mem_{header} \leq C \cdot (W_{flow} + D \cdot W_{pkt}) \tag{4.13}$$

With this model, we can compare the memory requirements of a header cache and a metric cache. For a metric cache, the memory requirement is:

$$mem_{metric} = C \cdot (W_{flow} + W_{stats}) \tag{4.14}$$

Here, W_{stats} is a fixed-width buffer that stores state necessary to compute and track streaming flow statistics.

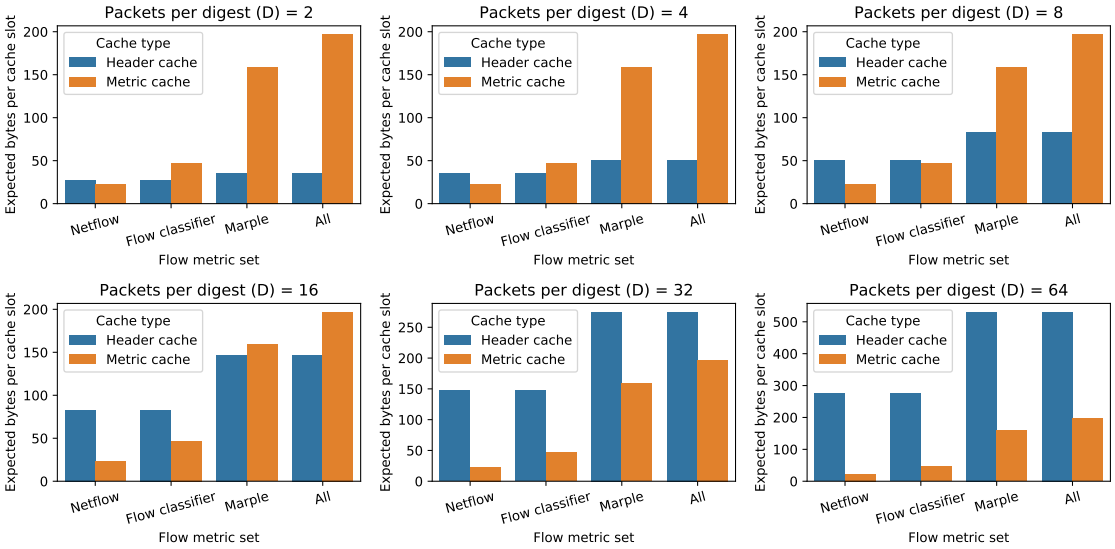

Figure 4.14: Per-slot memory requirements of metric and dynamically allocating header caches in telemetry systems that measure different sets of statistics.

Figure 4.14 analyzes memory requirements as D and the set of telemetry metrics vary. Table 4.2 summarizes the metric sets we use.

- *Basic Netflow* metrics include flow duration, packet count, and byte count. These metrics are supported by legacy Netflow switches.

Metric Set	Number of metrics	Per-flow state (W_{stats})	Header size (W_{pkt})
Basic Netflow	3	80b	32b
Flow classifier	7	272b	32b
Marple metrics	6	1168b	64b
All metrics	14	1472b	64b

Table 4.2: Flow metric sets with sizes of required per-flow state (for a metric cache) and header fields (for a header cache), in bits.

- *Flow classifier metrics* include average, standard deviation, and maximum packet size and inter-arrival times of each flow, plus the sizes of the first 8 packets in each flow. These metrics support systems that classify flows by application [94].

- *Marple metrics* include a subset of the metrics introduced in other recent telemetry work [11]: per-flow packet and connection initialization counts, average inter-arrival times, flowlet size histograms, and counters for out-of-order and non-monotonically increasing TCP sequence numbers.

Figure 4.14 illustrates that, surprisingly, a header cache can use *less* memory than a metric cache when $D \cdot W_{pkt} < W_{metric}$. For example, in an Internet workload D is typically under 4. If the telemetry system needs to support all 14 tested metrics, a header cache would use around 1/4 as much memory as a metric cache.

In datacenter workloads with small caches, D is between 8-32. In this range, metric and header cache memory requirements are within a factor of 2. In datacenter workloads with larger caches, D is higher and metric caches have the advantage, especially when the set of supported metrics is small (e.g., the *Netflow* or *Flow classifier* sets).

Static memory allocation. Fully dynamic memory allocation is efficient, but not possible to implement in today's line-rate reconfigurable ASICs [17]. The simplest alternative is to preallocate a static number of packet header buffers for each cache

slot, as done by the basic implementation of `*Flow`. When a slot's fixed-width buffer overflows, it must flush its current contents to the backend telemetry processor via a digest, to make room for new packets.

We can model the effect of slot overflows on the packet to digest ratio by extending the models for metric caches (Section 4.1). The core idea is to adjust the equation for $p_{hit}[f]$, the hit rate of flow f, to account for overflows.

First, we calculate the probability of observing each digest length k for flow f using $p_{hit}[f]$. A digest of length k is generated when there is a single miss followed by k hits.

$$P[D == k] \approx (1 - p_{hit}[f]) \cdot p_{hit}[f]^k \tag{4.15}$$

We then calculate D', the expected packets per digest when accounting for overflows. In a statically allocated cache with W packet buffers per slot, each digest of length k corresponds to $\lfloor \frac{k}{W} \rfloor$ overflow digests of length W and 1 collision digest of length $k \bmod W$. We then calculate the average over all k values likely to occur, i.e., $1 < k < 10000$.

$$D' = \frac{\sum_{k=1}^{N} P[D == k] \cdot (W \cdot \lfloor \frac{k}{W} \rfloor + k \bmod W)}{N} \tag{4.16}$$

Finally, we convert D' into a hit probability p'_{hit} that accounts for overflows. p'_{hit} is then used in place of p_{hit} in the model equations to calculate overall cache hit rate.

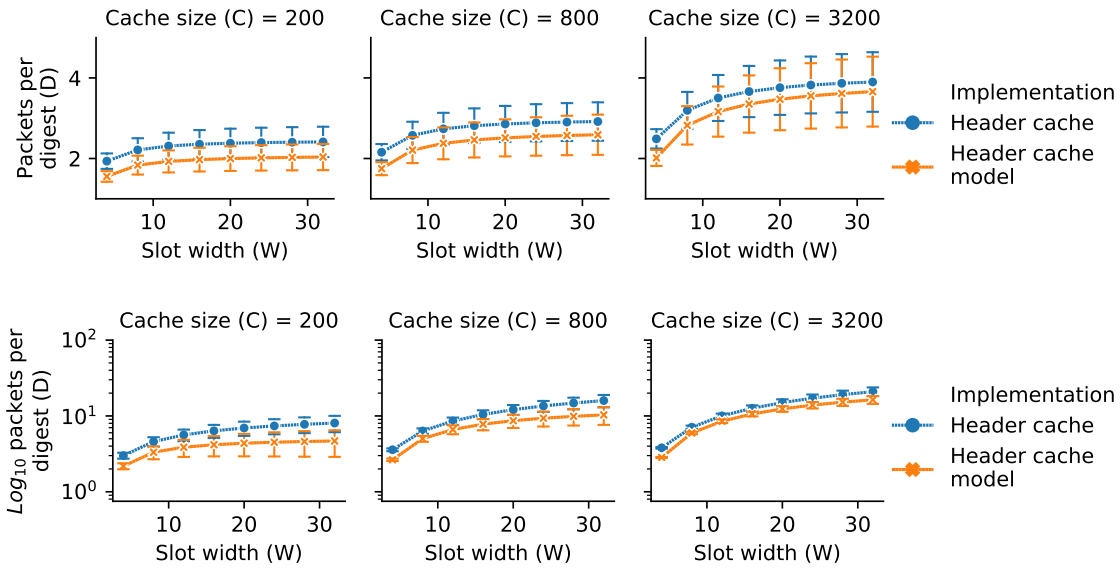

Figure 4.15: Packets per digest (D) predicted by the parametric model of a header cache with fixed-allocation. Empirical results from `*Flow` (*implementation*) shown for comparison. The top plots show Internet workloads and the bottom plots show datacenter workloads. Note that datacenter plots use log-scale Y axes.

Validation. Figure 4.15 shows model predictions as W and C vary for the Internet and datacenter workloads, with measurements from `*Flow` as a baseline. The model is accurate and conservative, like the underlying parametric model from Section 4.1. Qualitatively, the model captures the important trends: (1) D increases logarithmically with slot width; (2) D is an order of magnitude higher in data center workloads versus Internet workloads; (3) variance is higher in Internet workloads.

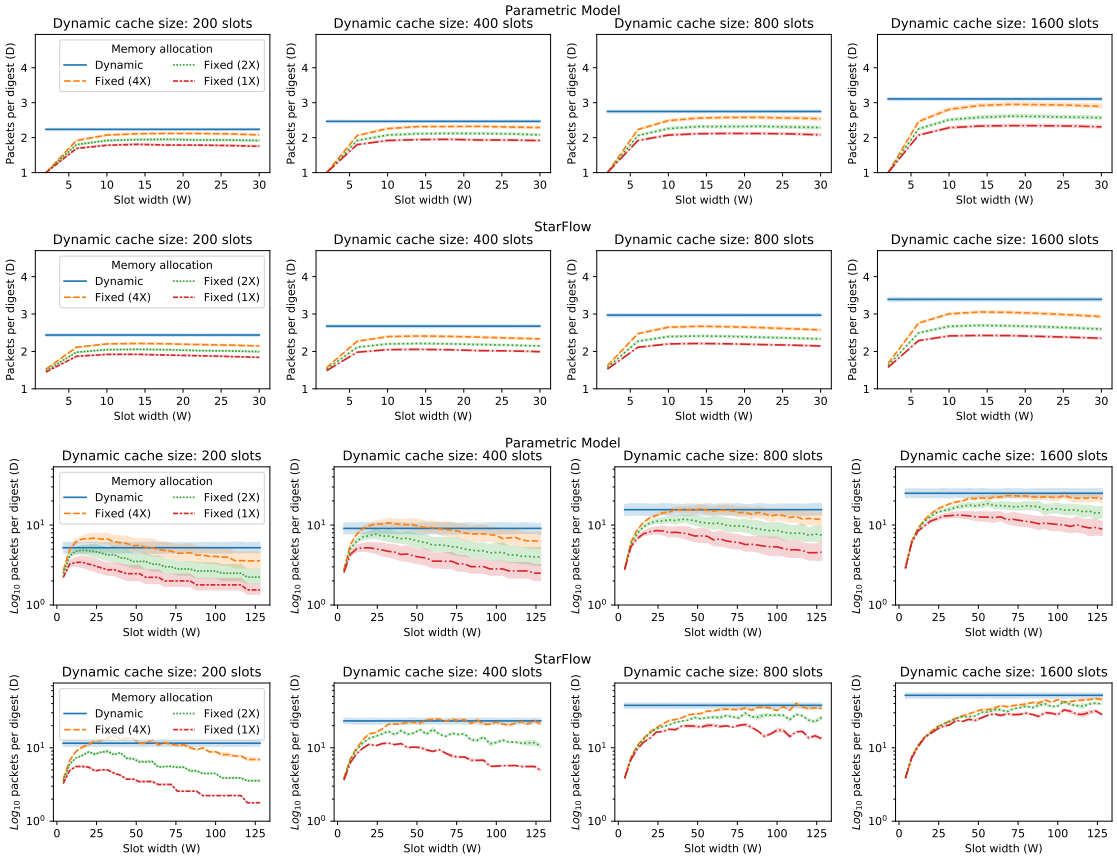

Figure 4.16: Average packets per digest (D) for statically allocated header caches using 1X, 2X, and 4X as much memory as a dynamically allocating header cache. The top 2 plots summarize Internet workloads while the bottom 2 plots summarize datacenter workloads. Note the log-scaled Y axis and larger X axis range in the datacenter plots.

Analysis. Using the model, we analyze the memory overhead of static allocation and D's sensitivity to W. Figure 4.16 compares the packet to digest ratio with static and dynamic allocation. Here, the dynamic cache is configured with C_{dyn} slots. For the static cache, we vary W and select the C_{sta} that produces the highest D value while keeping total memory utilization less than a threshold: either 1X, 2X, or 4X of the dynamic cache's average memory utilization.

Both the model and the system show similar trends. The left side of each plot is low because overflows are frequent when slots are narrow. There is a wide range of W values in the middle of each line where D is near optimal. In this range, a statically

allocated cache can achieve nearly the same D as a dynamic cache if given around 4X as much memory. The wide range of near-optimal W values also shows that statically allocated header caches are not highly sensitive to W as a tuning parameter.

As W increases beyond the optimal range, D eventually decreases. This is because a wider cache must have fewer slots to fit into a fixed memory budget. Reducing the number of slots increases the frequency of evictions due to capacity and hash collisions. The effect is stronger in the datacenter traces, where the benefit of additional cache slots is more significant (Figure 4.6).

Figure 4.16 also shows that large caches in datacenter scenarios perform best when slots are extremely wide, i.e., with capacity for more than 50 packet header buffers per slot. This is beyond the capacity of today's line rate switch datapaths. For example, the Barefoot Tofino supports 48 32-bit memory operations per packet (4 operations in each of 12 stages). Future datapaths could better support header caches by adding more stages, more parallel memory blocks in each stage, or wider memory buses (so multiple records could fit in the same word). Prior work suggests that these extensions are practical and would not increase cost (chip area) significantly [17, 18].

Comparison to Metric Caches. Static header caches generally require more memory than metric caches because of the 4X overhead of static allocation shown in Figure 4.16. Based on Figure 4.14, which compares metric and dynamically allocating header caches, we see that this 4X overhead would put a static allocating header cache above a metric cache in all scenarios except for the "Marple" and "All" metric sets when $D \leq 4$.

Datapath Cache Summary

In summary, a datapath telemetry cache summarizes packets into digests to reduce the workload of telemetry processing servers. This operation requires per flow state, and it is somewhat surprising that a datapath cache can provide a significant benefit

given its limited memory and the large number of flows.

Analysis with simple models explain that the effectiveness has different causes in Internet and datacenter workloads. For Internet workloads, effectiveness is due to a combination of *source* and *burst* locality in the underlying network traffic. We demonstrate the causal relationship by deriving statistics to quantify each axis of locality and building a simple generative model that accurately predicts cache performance based on the locality statistics. With the model, we show that a telemetry cache exploits both axes of locality to achieve high performance in a wide range of traffic workloads.

A telemetry cache is more effective in datacenter workloads versus Internet workloads because there are orders of magnitude fewer active flows. In such scenarios, hash table associativity is important for achieving a high hit rate, but associative hash tables are too complex for implementation in a line rate switch datapath. Models and analysis guide us to a simple multi-layer mapping scheme that approximates associativity to improve cache performance by a factor of up to $6X$ and can be implemented in highly restricted datapaths. The performance of 4-layer and fully associative caches converge rapidly when the cache can be sized large enough to store more than 2X the number of concurrently active flows.

Packet header caches are more flexible than flow metric caches. Surprisingly, our models and analysis show that they can also require less memory when dynamic memory allocation is supported. Static allocation, i.e., for a restrictive line rate datapath implementation, has around a 4X memory overhead compared with dynamic allocation. For larger caches, our analysis shows that optimal performance requires wider slots than today's line rate datapaths can support. This motivates future hardware with longer pipelines or wider pipeline stages.

4.2 Modeling the Telemetry Store

Flow digests from telemetry caches stream to the backend of a telemetry system, a telemetry processor (or store) that computes additional metrics and summarizes flows over longer periods of time. Telemetry stores are typically implemented in software for general purpose processors and, unlike telemetry caches, are not line rate. Their performance depends on algorithm and implementation choices, as well as specifics of the workload. In this section, we model and analyze these relationships.

A High Level Model

The main performance goal of a telemetry store is to minimize $\overline{t_{pkt}}$, the average amount of time spent processing each packet. A lower $\overline{t_{pkt}}$ means that the store can achieve a higher throughput without requiring additional resources. We can model $\overline{t_{pkt}}$ based on $\overline{D}$, the average number of packets per digest, and the cost of the three high level operations that a telemetry store must perform: hash computation, flow lookup, and metric calculation.

$$\overline{t_{pkt}} = \frac{\overline{t_{hash}} + \overline{t_{lookup}}}{D} + \frac{\overline{t_{calc}}}{D'} \tag{4.17}$$

$$D' = \begin{cases} D & \text{if using a metric cache} \\ 1 & \text{if using a header cache} \end{cases} \tag{4.18}$$

In this equation:

- $\overline{t_{hash}}$ is the per-digest time to calculate a hash of the flow's key;
- $\overline{t_{lookup}}$ is the per-digest time to find a record of the flow's prior state, if it exists, or insert a new record if it does not;
- D is the *packet to digest ratio* of the telemetry cache;

- and $\overline{t_{calc}}$ is the per-digest (for telemetry with a metric cache) or per-packet (for telemetry with a header cache) time to calculate flow metrics.

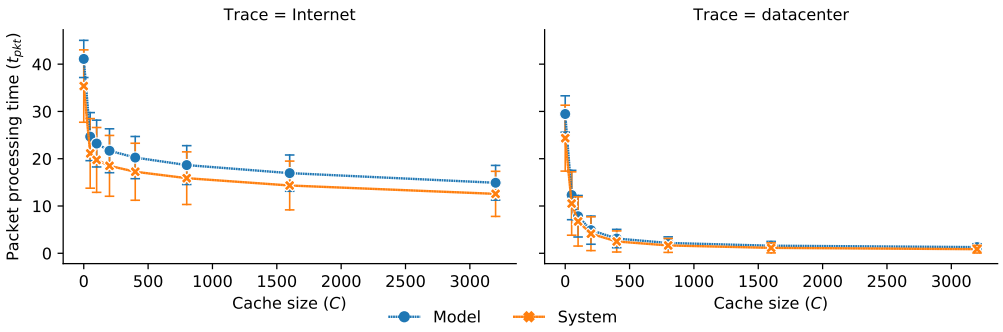

Figure 4.17: Telemetry store performance (`TurboFlow`), modeled and observed, as telemetry cache size varies. $\overline{t_{pkt}}$ measures in-memory processing time on 1 core, lower is better. Markers and bars show averages and standard deviations over 40 trials.

Validation with `TurboFlow`. To validate the model, we instrumented `TurboFlow` to measure all t values using the x86 `RDTSCP` instruction. Figure 4.17 compares measured and modeled $\overline{t_{pkt}}$ for the Internet and datacenter traces as the size of the underlying telemetry cache varies. Using measured inputs, the simple model (Equation 4.18) predicts processing times within 1 standard deviation of observed times for all evaluated cache sizes. Like the measured system, it shows that a telemetry cache reduces $\overline{t_{pkt}}$ significantly, by up to 2.8X (for the Internet traces) and over 20X in the datacenter traces.

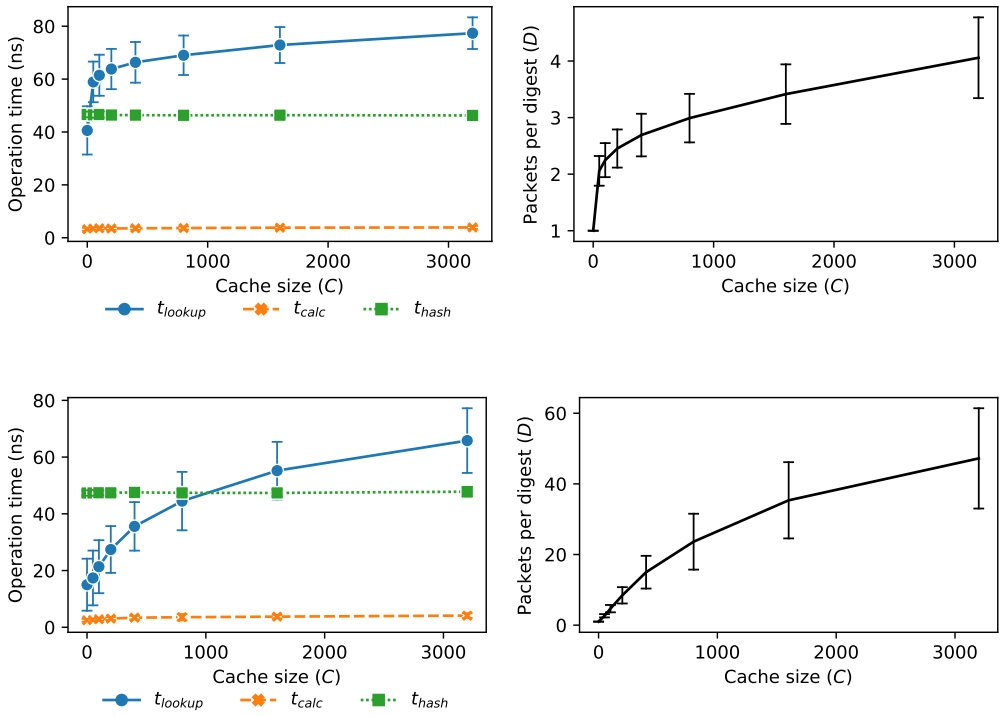

Figure 4.18: Measured parameters of Equation 4.18 for Internet (top) and datacenter (bottom) traces, as telemetry cache size varies. Markers show average and bars show standard deviation over 40 trials.

Using the model, we can explain the $\overline{t_{pkt}}$ performance curve. Figure 4.18 plots all variables in Equation 4.18. As the size of the telemetry cache increases so does D, the packet to digest ratio. This causes $\overline{t_{pkt}}$ to decrease. However, as the size of the telemetry cache increases, $\overline{t_{lookup}}$ also increases, which reduces the benefit of a higher D. In the remainder of this section, we model $\overline{t_{lookup}}$ and $\overline{t_{calc}}$ to better understand when and why they change.

Lookup time ($\overline{t_{lookup}}$)

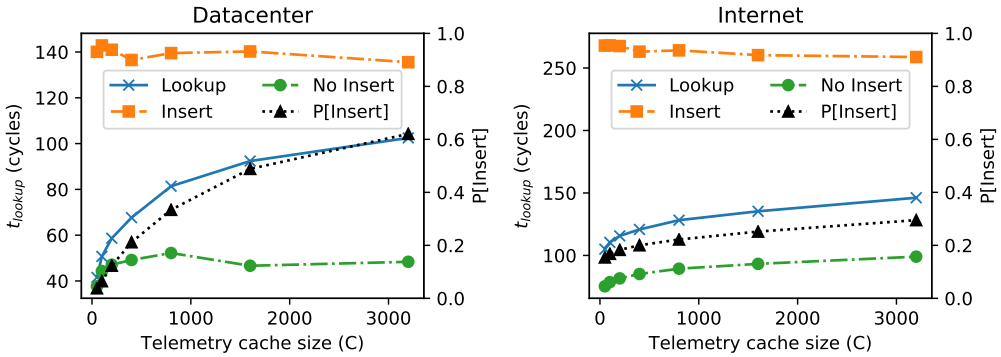

Figure 4.19: Flow lookup times in a telemetry store as cache size varies. Each point is the averge over 40 trials. The solid line shows overall average, dashed lines show average depending on whether or not an insert operation was required, and the dotted line shows probability of an insert operation.

Why does $\overline{t_{lookup}}$ increase with the telemetry cache's size? At a high level, there are two reasons. First, as Figure 4.19 shows, flow lookups are more expensive when the store needs to insert a new record, i.e., for the first digest of each flow. As cache size and D increase, so does the relative frequency of insert operations. This causes $\overline{t_{lookup}}$ to increase in both the datacenter and Internet traces.

Additionally, as Figure 4.19(right) shows, the average lookup time can increase even when there are no insert operations. This is because a larger telemetry cache consumes more of the temporal locality in the underlying stream of packet measurements, reducing the benefit of the cache hierarchy in the telemetry store's CPU.

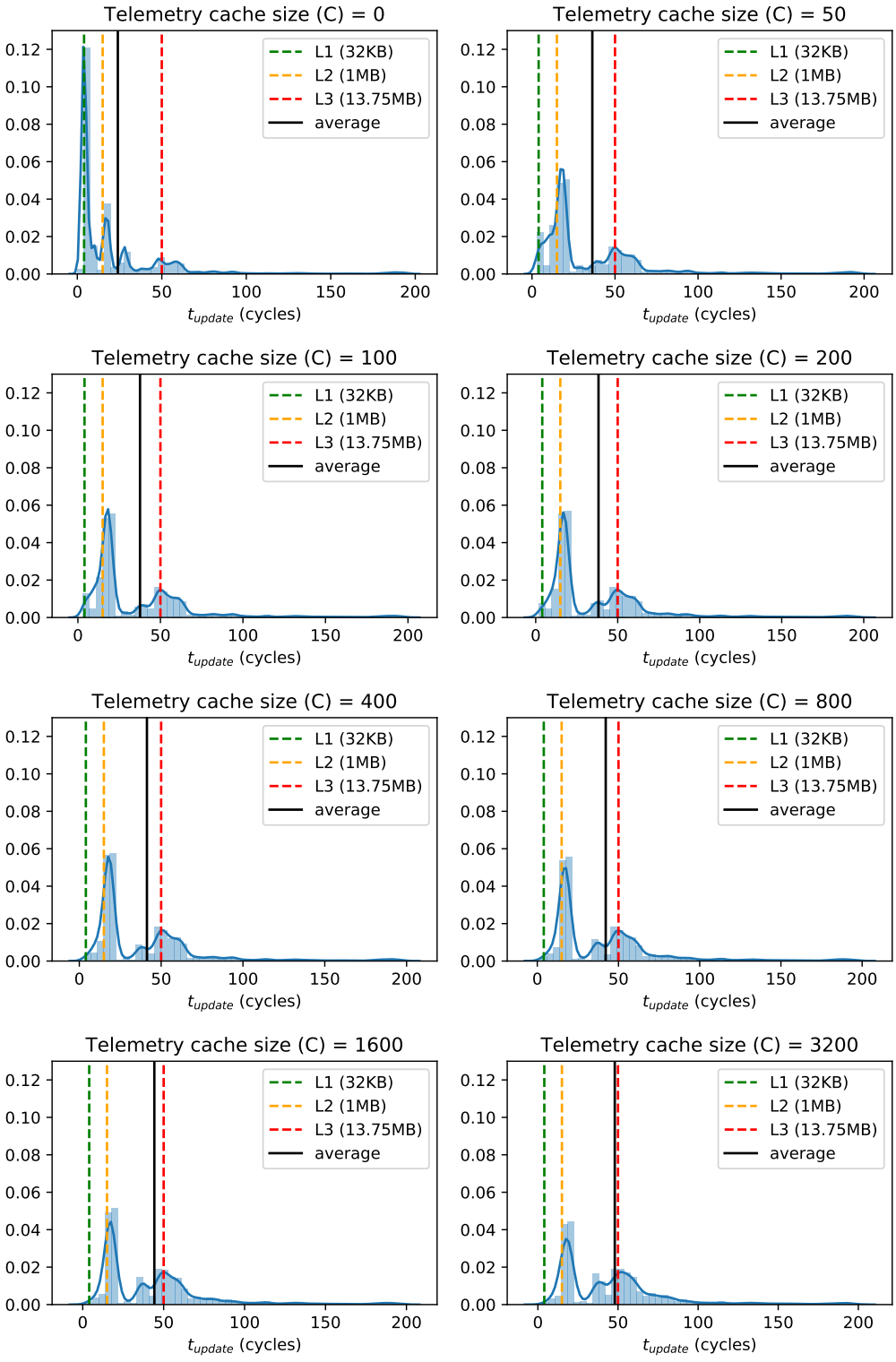

Figure 4.20: The distribution of $\overline{t_{lookup}}$ values at a telemetry store as the telemetry cache size varies. Dashed lines show access times for caches in the telemetry store's CPU.

Figure 4.20 illustrates the effect by plotting the distribution of all t_{lookup} values measured in 40 1-second trials of `TurboFlow` on the Internet router trace. Without a telemetry cache, a significant fraction of the lookups are for recently accessed flow records still in the server's fast (>4 cycle access time) but small (32KB) level 1 cache. As the size of the telemetry cache increases, load shifts from the level 1 cache to the larger and slower level 2 and 3 caches. For larger workloads, where the set of active flows is too large to fit in the level 3 cache, we would also expect a fourth concentration of points centered around the access time of main memory (around 150 ns).

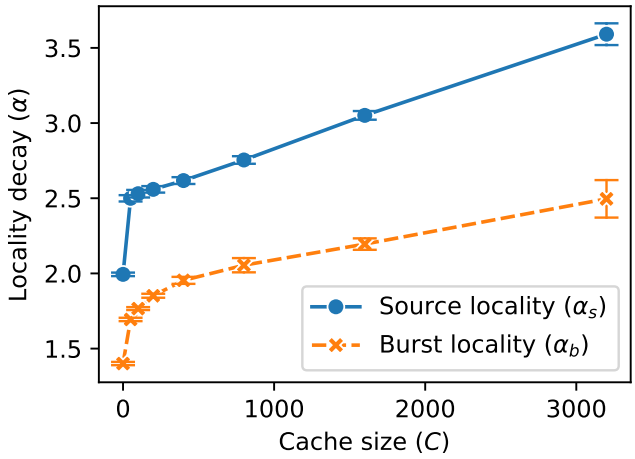

Figure 4.21: Locality statistics in the digest streams exported by `TurboFlow` running on the Internet router traces. Higher values signify less locality. Markers and bars show average and 99% confidence intervals over 40 trials.

We can quantify the loss of locality using the locality statistics introduced in Section 4.1. Recall that there are two statistics, α_s and α_b, that quantify locality in a multi-flow stream. α_s measures the decay of source locality across flows, a higher value indicates fewer heavy hitter flows and thus less locality. α_b quantifies the decay of burst locality across flows, a higher value indicates less locality due to smoother flows with more uniform inter-arrival times.

Figure 4.21 show how the locality statistics of a digest stream change with teleme-

try cache size (C) for the Internet traces. Adding a small telemetry cache increases both locality statistics by approximately 25% and both statistics increase logarithmically with C.

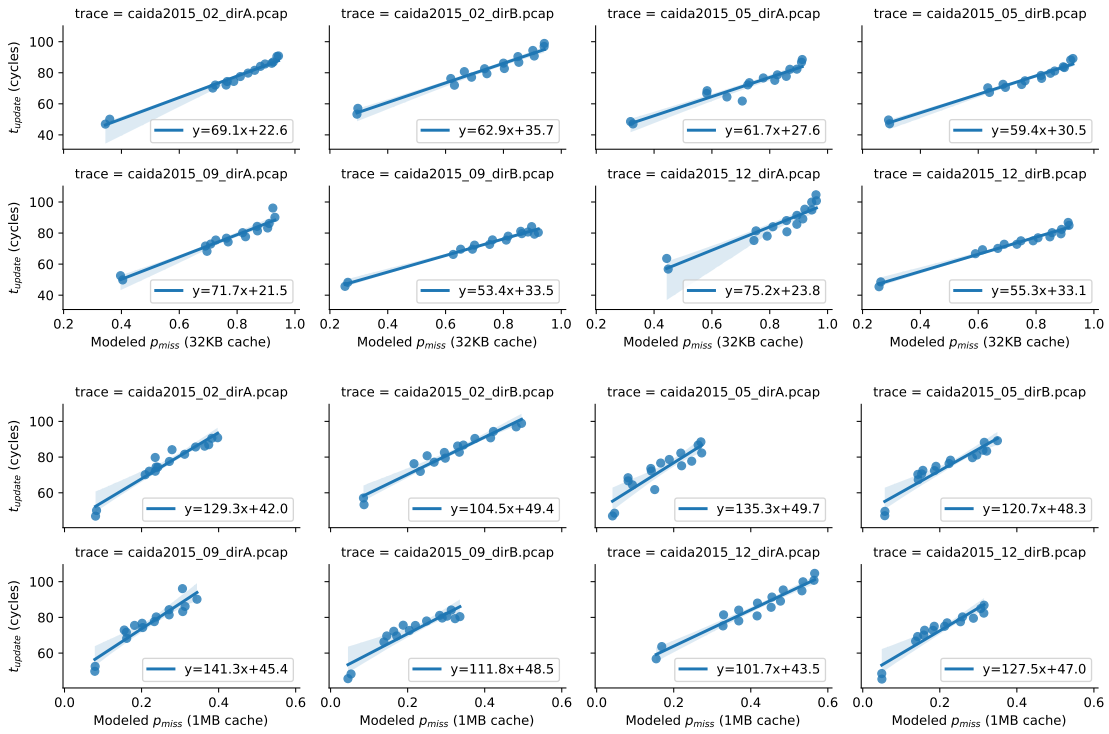

Figure 4.22: $\overline{t_{lookup}}$ (with no insertions) in a telemetry store (`TurboFlow`) versus the modeled miss rate (p_{miss}) of 32KB and 1MB telemetry caches (top and bottom). Each point shows the average over 10 1-second trials with a fixed telemetry cache size.

We can relate change in locality statistics to the change in $\overline{t_{lookup}}$ due to expected cache misses by plugging the locality statistics into our parametric cache model (Section 4.1). Figure 4.22 plots $\overline{t_{lookup}}$ versus the modeled miss rate of telemetry caches sized for the layer 1 and layer 2 CPU caches. There is a strong linear relationship between modeled miss rate and lookup time, as expected from Figure 4.20. Based on the coefficients in the linear best fit equations, we can also see that a L2 miss is around 2X more expensive than a L1 miss, on average. This corresponds to the difference in L1 versus L2 cache access times (around 4 ns versus around 7 ns for our

Intel Xeon Silver 4114).

Calculation time ($\overline{t_{calc}}$)

$\overline{t_{calc}}$ is the average amount of time that the telemetry store spends on updating flow metrics to account for each packet. $\overline{t_{calc}}$ depends on the set of flow metrics being used. Metrics that are more computationally intensive will take longer to calculate, as will larger sets of metrics.

Additionally, even for the same metric set, $\overline{t_{calc}}$ depends on whether the underlying telemetry cache tracks metrics or headers. With a metric cache, $\overline{t_{calc}}$ is a per-digest cost that is amortized over all the packets summarized in each digest. With a header cache, $\overline{t_{calc}}$ is a per-packet cost since the store must iterate over all the packet header records in each digest to calculate statistics.

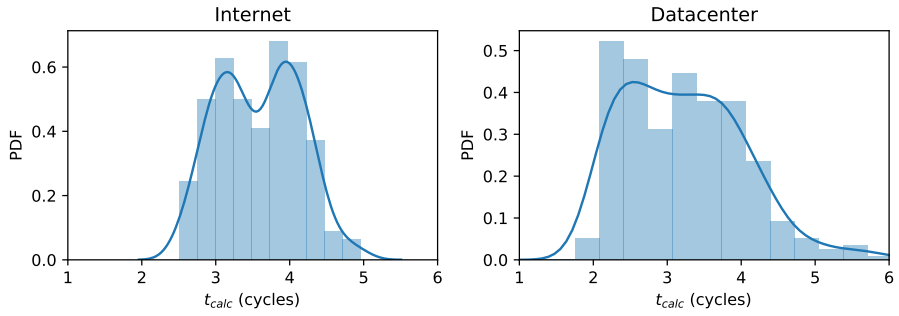

Figure 4.23: $\overline{t_{calc}}$ for a telemetry system that calculates 3 simple counters (packet counter, byte counter, and digest counter) from a stream of digests emitted by a header cache (`*Flow`).

Simple Metrics. With simple metrics, such as the counters and average statistics found in traditional Netflow record, $\overline{t_{calc}}$ is small and has little impact on overall $\overline{t_{pkt}}$. For example, Figure 4.23 shows the distribution of $\overline{t_{calc}}$ cycle times when computing three simple per-flow counters. The average $\overline{t_{calc}}$ time is around 4 cycles.

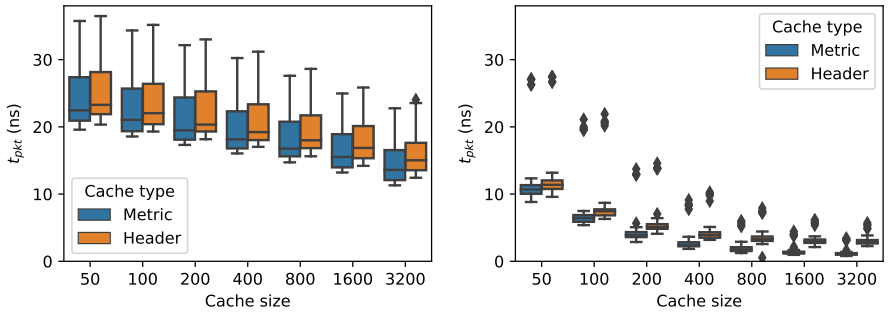

Figure 4.24: $\overline{t_{pkt}}$ versus telemetry cache size when generating simple Netflow records with metric and header caches. Boxes show quartiles, whiskers show 90th percentiles, and dots show outliers.

As Figure 4.24 shows, for these simple metrics, telemetry system performance is similar with both types of caches. The difference is under 10% for all Internet trials and most datacenter trials. It is largest in data center trials with large caches (the rightmost bars in Figure 4.24). At this point, the average absolute difference between a metric and header cache is 2ns per packet.

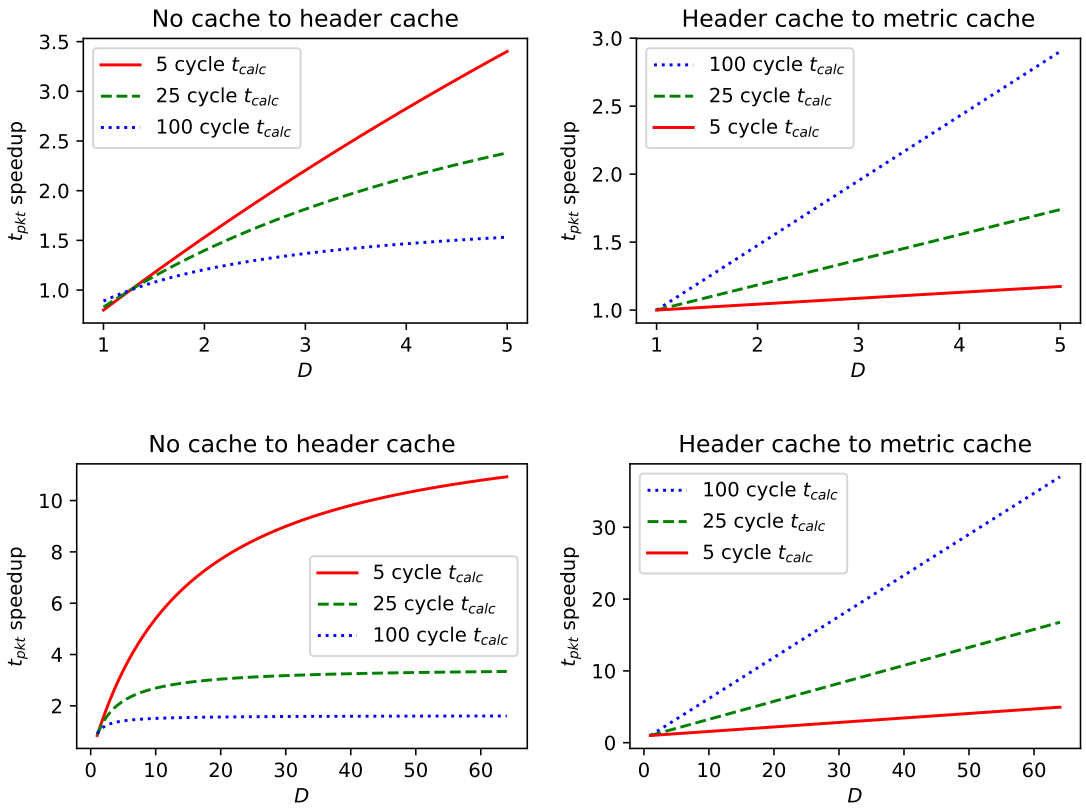

Figure 4.25: $\overline{t_{pkt}}$ speedup (higher is better) of header and metric telemetry caches as D (packets per digest) and $\overline{t_{calc}}$ vary, in the Internet (top) and datacenter (bottom, with log-scaled Y axis) traces.

Speedup. Figure 4.25 characterizes the relationship between D, $\overline{t_{calc}}$, and $\overline{t_{pkt}}$ by measuring $\overline{t_{pkt}}$ speedup: the reduction in $\overline{t_{pkt}}$ when changing from either no cache to a header cache, or from a header cache to a metric cache. When D is low (under 5, as typically observed in the Internet traces), a header cache provides significant (over 2X) speedup when $\overline{t_{calc}}$ is less than approximately 25 cycles. The metric cache only provides significant speedup over the header cache when $\overline{t_{calc}}$ is high enough to dominate the other terms in Equation 4.18, e.g., near 100 cycles.

When D is high (up to 64, as observed in the datacenter traces with large cache sizes), the benefit of a metric cache increases dramatically, providing over a 10X speedup when $\overline{t_{calc}} < 25$ cycles and a 4X speedup even when $\overline{t_{calc}}$ is low. This can be

explained by Equation 4.18: when D is high, the first term of the equation becomes so small that *any* unamortized $\overline{t_{calc}}$ dominates overall $\overline{t_{pkt}}$.

Datapath Hardware Limitations. Although metric caches provide more benefit when metrics are more computationally expensive, they cannot support arbitrarily complex functions. This limits the scenarios in which they can be used. For example, metrics that require floating-point operations or multiple passes over packet records cannot be calculated in current switch datapaths [16, 11].

To understand the edge cases of today's hardware, we consider three calculation functions that are at the limit of what the Tofino datapath can support. Each function represents flow metric calculation that is complex in different ways.

- The ***compute bound*** function sums 128 fields of packet metadata, performing 1154 parallel addition operations and accumulating the results into a single integer. This function consumes all of the Tofino's available packet metadata memory.
- The ***memory bound*** function increments 48 fields in memory that persists across packets. It consumes all of the Tofino's available SRAM ports.
- The ***code path bound*** function performs 24 sequential additions. This function requires processing in each of the Tofino's 24 stages.

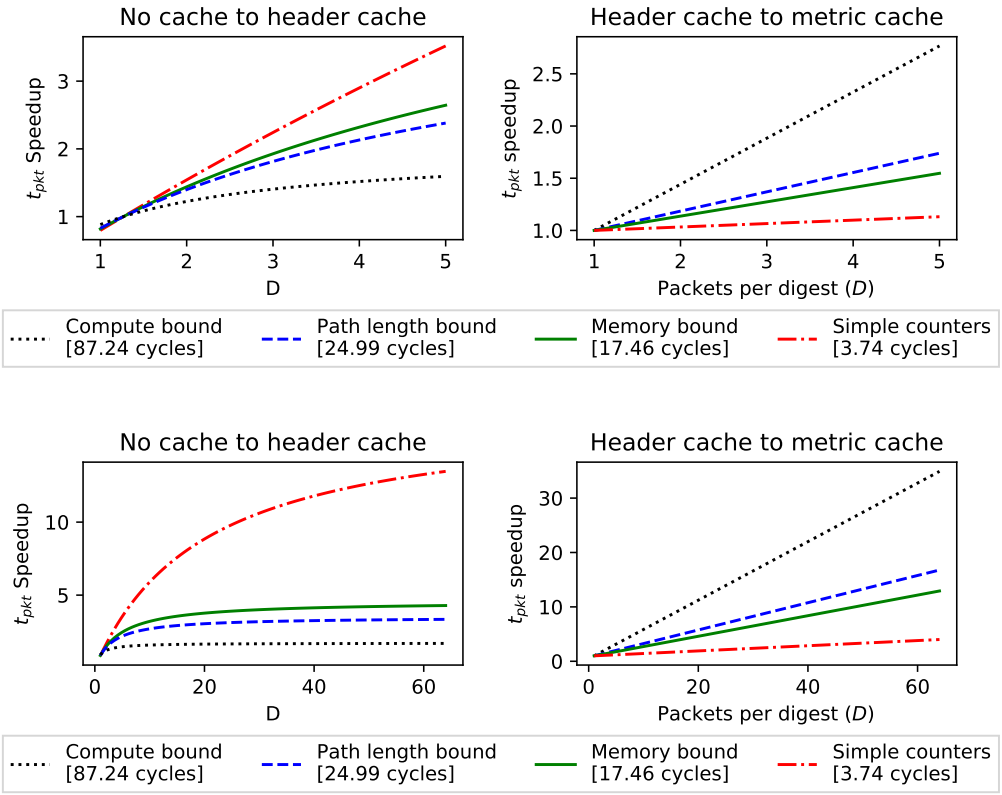

Figure 4.26: $\overline{t_{pkt}}$ speedup of header (left) and metric (right) caches with functions that fully utilize different classes of resources in a Barefoot Tofino datapath. The legend reports the average $\overline{t_{calc}}$ of vectorized implementations on an Intel Xeon Silver 4114 with 512-bit vector operations (SSE 4.2).[5]

Figure 4.26 plots the speedup of a header and metric cache for the three functions. Compute bound functions that can take full advantage of the Tofino's high degree of instruction level parallelism benefit significantly from offload to the metric cache. Memory bound and complex functions can benefit from offloading, but only when D is high.

[5]We compiled with gcc 9 and the following flags: `-Ofast -funroll-loops -mprefer-vector-width=512 -march=skylake-avx512 -fopt-info-vec-optimized -mavx512f -mavx512dq -mavx512bw -mavx512vbmi -mavx512vbmi2 -mavx512vl`.

Summary

A telemetry store aggregates digests from telemetry caches into complete flow records. Unlike telemetry caches, a store does not run at line rate. This makes $\overline{t_{pkt}}$, the average per-packet processing time, its most important performance metric. To the first order, we can model $\overline{t_{pkt}}$ as the sum of two terms (Equation 4.18): $\overline{t_{lookup}}$, the time spent finding the flow record corresponding to each digest; and $\overline{t_{calc}}$, the time spent calculating flow metrics.

A telemetry cache reduces $\overline{t_{lookup}}$ by amortizing its cost across the D packets summarized in a digest. This can reduce $\overline{t_{pkt}}$ significantly, by around 2-3X in Internet scenarios or up to 20X in datacenter scenarios 4.17. However, the benefit does not scale linearly with D because of two factors. First, flow insertion operations, which are more expensive than lookups of existing records, become more common as D increases (Figure 4.19). Second, a telemetry cache consumes locality by transforming a packet stream into a digest stream, which reduces benefit of the cache hierarchy in the telemetry store's CPU (Figure 4.20).

Metric caches, which emit digests containing statistics instead of header records, also amortize $\overline{t_{calc}}$ by D. The additional speedup gained by deploying a metric cache instead of a header cache (`TurboFlow` instead of `*Flow`), depends on the metrics and the achievable values of D. A metric cache can provide over a 30X speedup versus a header cache when D is large and $\overline{t_{calc}}$ is high. However, in scenarios where D is low or the telemetry processor can calculate desired metrics quickly, the speedup is much lower. For simple metrics, such as averages, counters, and other streaming measurements, the $\overline{t_{calc}}$ speedup of a metric cache is under a factor of 2 for most achievable D values.

4.3 Conclusion

Hybrid telemetry systems split packet aggregation and metric calculation across caches in switch datapaths and stores on commodity servers. They balance performance and flexibility, but are architecturally complex, which makes it difficult to understand when and why hybridization is beneficial. As a solution, this chapter developed a framework for analyzing the performance of hybrid telemetry systems based on simple but accurate models of both telemetry caches and telemetry stores.

The cache models revealed the causal relationship between traffic characteristics (source locality, burst locality, and number of concurrent flows) and cache effectiveness. Model-based analysis showed that the high performance observed in the benchmarks from prior chapters are a general phenomena that occur for a wide range of traffic workload parameters. The model predicted that a lack of associativity was a major performance limitation in datacenter workloads. This was validated with a novel multi-hash data structure that approximated associativity to improve cache effectiveness by up to a factor of $6X$.

The store model related overall system performance to the size of the underlying telemetry cache and the cost of three simple high level operations: key hashing, flow lookup, and metric calculation. Analysis of the lookup term showed that the workload reduction of a larger telemetry cache is partially offset by increased per-digest lookup times. This was due to more frequent insert operations and the consumption of locality that would otherwise be exploited by the telemetry store's CPU.

The store model enabled comparison of telemetry system performance using metric versus header caches. Analysis showed that a metric cache provided a significant speedup over a header cache in two scenarios: when the degree of aggregation was high or when statistic calculation was compute intensive.

Chapter 5

Related Work

This chapter summarizes areas of research that are related to the systems introduced by this thesis.

5.1 Hardware Accelerated Telemetry

This work builds on other recent hardware accelerated telemetry systems [11, 15] with different design goals. These systems are designed for periodic measurement tasks, e.g., a network administrator manually debugging an incast by querying specific switches for statistics about certain flows. On the other hand, TurboFlow and *Flow are designed for high coverage and always-on monitoring. The systems meet these more aggressive performance demands because hardware and software is integrated and co-specialized, which reduces overhead without sacrificing flexibility. Marple and FlowRadar, which are not designed with these principles, rely on external servers to post process data exported from switch hardware, which adds overhead that makes high coverage monitoring cost prohibitive.

5.2 Query Refinement

This thesis is complementary to concurrent work on network query refinement [114, 16]. Refinement can be applied either at runtime or at compile time. When applied at runtime, the data plane of a network streams an increasingly selective flow of packets to a software processor. The refinement reduces the overhead of monitoring packet payloads, but also loses information about all the flows filtered out. TurboFlow and *Flow account for all packets in exported data and are lightweight enough to run at all times, but do not provide information about packet payloads. A natural way of combining the systems would be to leverage data from TurboFlow and *Flow as context to guide refinement of packet queries.

Compile time refinement is an optimization that can reduce the memory required to calculate certain statistics without losing information [16]. TurboFlow and *Flow could benefit, but are not as sensitive to memory limitations due to integration of general purpose CPUs with large memory banks.

5.3 CPU Assisted Forwarding

Several prior systems have proposed coupling traditional fixed function FEs with switch CPUs, using the CPUs for caching forwarding rules [115, 116, 117], more flexible packet processing [118, 119, 120], or counter processing [121]. In these systems the CPU has a fixed interface to the FE, similar to OpenFlow [122], that allows it to send and receive packets, poll counters, and install forwarding rules. The fixed interface and high cost of forwarding rule installation are obstacles to using these systems for FR generation. The only way for the CPU to offload FR generation work to the FE is by installing per-flow forwarding rules and periodically polling their counter values. This strategy leaves the CPU with too much work because forwarding rule installation rates are much lower than flow arrival rates, e.g., 300-1000 per second [117], compared to >10,000 new flows per second for a single 10

Gb/s link [123]. TurboFlow and *Flow leverage the increased flexibility of PFEs to implement a custom mFR based interface between the PFE and CPU that allows the work to be partitioned at a finer granularity for better PFE utilization.

5.4 Packet Processor Specialization

TurboFlow related to work that specializes other elements of packet processors including lookup tables [124, 125], key value stores [126], software switches [127], and sampled FR generation [128]. There is overlap in the optimizations that all of these systems use, e.g., batching is generally effective. However, TurboFlow is the first to propose and evaluate the specific hash table optimizations described in Section 2.5 for the task of FR generation with commodity switch CPUs, which are much less powerful than server CPUs.

5.5 Energy Savings

A primary goal of this thesis is reducing the resource cost of network monitoring, including the axis of energy usage. Many other works have demonstrate the practical importance and challenge of reducing power consumption with, for example, energy saving load balancers [51, 56] and architectures [57]. TurboFlow and *Flow represent a complimentary way to reduce power consumption in telemetry capable networks, independent of routing or architecture.

Additionally, telemetry is critical for systems that seek to tune the network to optimize for power usage [51, 56]. The systems presented in this thesis can serve as a platform to support the necessary monitoring at low cost. Wider support for flexible network telemetry with low overhead can also open the door to more sophisticated control algorithms that further improve efficiency.

Chapter 6

Conclusion

Telemetry is a critical feature in high speed data planes because it enables a wide range of applications that improve the security, performance, and management of networks. Due to the demands of their operating environments and the applications that they support, telemetry systems must balance performance and flexibility requirements with inherent tensions. Thus far, it has proven challenging to meet fundamental performance demands, e.g., throughput and coverage, let alone balance them with application-driven flexibility requirements.

A hybrid approach can resolve these tensions by exploiting general properties of telemetry workloads and hardware heterogeneity. As this dissertation showed, the competing performance demands of throughput, coverage, and cost can all be met simultaneously by organizing diverse memory and compute hardware to exploit workload locality and small packet feature sets. These insights enable performance-oriented telemetry systems, e.g. `TurboFlow`, that support full coverage measurement at multi-terabit rates.

Hybridization also allows telemetry systems to support a range of important flexibility requirements while still achieving high performance. The key is to carefully divide work between restrictive line-rate processors and general-purpose processors with inherent bottlenecks. A novel intermediate format for telemetry data, introduced

in this dissertation, allows a line-rate front-end processor to alleviate the bottlenecks of a general-purpose back-end processor without limiting its flexibility. The front-end extracts and gathers relevant packet data, while the back-end performs arbitrary metric calculation on the gathered data. This division of work enables balanced hybrid telemetry systems, e.g. `*Flow`, that achieve high performance while also meeting application-motivated flexibility goals.

These ideas have been validated by working implementations of `TurboFlow` and `*Flow` on readily available commodity hardware. Benchmarks on Internet and data-center workloads demonstrate the capability to operate at terabit rates while meeting all other motivating performance and flexibility goals.

To generalize the systems and results, this dissertation introduced simple but accurate performance models for telemetry systems. These models relate performance to high level parameters characterizing traffic workloads, measurement tasks, and heterogeneous hardware. This framework of models explains the principles that make hybridization effective for telemetry, identify operational limits, and lay a foundation for future work.

Bibliography

[1] J. Steinberger, L. Schehlmann, S. Abt, and H. Baier, "Anomaly detection and mitigation at internet scale: A survey," in *IFIP International Conference on Autonomous Infrastructure, Management and Security*, pp. 49–60, Springer, 2013.

[2] C. Mills, D. Hirsh, and G. Ruth, "Internet accounting: background," tech. rep., 1991.

[3] K. C. Claffy, H.-W. Braun, and G. C. Polyzos, "A parameterizable methodology for internet traffic flow profiling," *IEEE Journal on selected areas in communications*, vol. 13, no. 8, pp. 1481–1494, 1995.

[4] C. So-In, "A survey of network traffic monitoring and analysis tools," *Cse 576m computer system analysis project, Washington University in St. Louis*, 2009.

[5] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of ip flow-based intrusion detection," *IEEE communications surveys & tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, vol. 7, pp. 19–19, 2010.

[7] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, p. 8, ACM, 2011.

[8] A. Kind, D. Gantenbein, and H. Etoh, "Relationship discovery with netflow to enable business-driven it management," in *Business-Driven IT Management, 2006. BDIM'06. The First IEEE/IFIP International Workshop on*, pp. 63–70, IEEE, 2006.

[9] K. Lakkaraju, W. Yurcik, and A. J. Lee, "Nvisionip: netflow visualizations of system state for security situational awareness," in *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pp. 65–72, ACM, 2004.

[10] W. Yurcik, "Visualizing netflows for security at line speed: The sift tool suite.," in *LISA*, pp. 169–176, 2005.

[11] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 85–98, ACM, 2017.

[12] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, pp. 50–60, ACM, 2005.

[13] L. Deri and N. SpA, "nprobe: an open source netflow probe for gigabit networks," in *TERENA Networking Conference*, 2003.

[14] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," tech. rep., 2001.

[15] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 311–324, 2016.

[16] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pp. 357–371, ACM, 2018.

[17] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 15–28, ACM, 2016.

[18] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 99–110, ACM, 2013.

[19] B. Networks, "Barefoot tofino." `https://www.barefootnetworks.com/technology/#tofino`.

[20] S. Goldberg and J. Rexford, "Security vulnerabilities and solutions for packet sampling," in *Sarnoff Symposium, 2007 IEEE*, pp. 1–7, IEEE, 2007.

[21] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the impact of sampling on netflow traffic classification," *Computer Networks*, vol. 55, no. 5, pp. 1083–1099, 2011.

[22] "Snort." http://www.snort.org/, 2013.

[23] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: an ensemble of autoencoders for online network intrusion detection," *arXiv preprint arXiv:1802.09089*, 2018.

[24] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, vol. 10, pp. 29–42, 2013.

[25] Cisco, "Cisco nexus 9200 platform switches architecture." `https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-737204.pdf`, 2016.

[26] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pp. 327–341, ACM, 2018.

[27] Cisco, "Cisco netflow generation appliance 3340 data sheet." `http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/netflow-generation-3000-series-appliances/data_sheet_c78-720958.html`, July 2015.

[28] Endace, "Endaceflow 4000 series netflow generators." `https://www.endace.com/endace-netflow-datasheet.pdf`, 2016.

[29] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "csamp: A system for network-wide flow monitoring.," in *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, vol. 8, pp. 233–246, 2008.

[30] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *ACM SIGCOMM Computer Communication Review*, vol. 34, pp. 245–256, ACM, 2004.

[31] R. Jain and S. Routhier, "Packet trains–measurements and a new model for computer network traffic," *IEEE journal on selected areas in Communications*, vol. 4, no. 6, pp. 986–995, 1986.

[32] M. Marchetti, F. Pierazzi, M. Colajanni, and A. Guido, "Analysis of high volumes of network traffic for advanced persistent threat detection," *Computer Networks*, vol. 109, pp. 127–141, 2016.

[33] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 123–137, ACM, 2015.

[34] D. Shan, F. Ren, P. Cheng, and R. Shu, "Micro-burst in data centers: Observations, implications, and applications," *arXiv preprint arXiv:1604.07621*, 2016.

[35] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, ACM, 2010.

[36] R. Philippe, "Cisco advantage series next generation data center flow telemetry," 2016.

[37] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: from packet capture to data analysis with netflow and ipfix," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[38] Edgecore, "Wedge 100bf-32x/65x switch datasheet." `https://www.edge-core.com/_upload/images/Wedge100BF-32X_65X_DS_R04_20180531.pdf`, May 2018.

[39] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Website fingerprinting at internet scale," in *Proceedings of the 23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*, 2016.

[40] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton, "Seeing through network-protocol obfuscation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 57–69, ACM, 2015.

[41] W. Lu and A. A. Ghorbani, "Network anomaly detection based on wavelet analysis," *EURASIP Journal on Advances in Signal Processing*, vol. 2009, p. 4, 2009.

[42] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: methods, systems and tools," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 303–336, 2014.

[43] B. Li, J. Springer, G. Bebis, and M. H. Gunes, "A survey of network flow applications," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567–581, 2013.

[44] Netronome, "Agilio cx intelligent server adapters agilio cx intelligent server adapters." `https://www.netronome.com/products/agilio-cx/`, 2018.

[45] Cisco, "Introduction to cisco ios netflow." `https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html`, 2012.

[46] B. Claise, B. Trammell, and P. Aitken, "Specification of the ip flow information export (ipfix) protocol for the exchange of flow information," tech. rep., 2013.

[47] Caida, "Trace statistics for caida passive oc48 and oc192 traces – 2015-2-19." `https://www.caida.org/data/passive/trace_stats/`, February 2015.

[48] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox: Modern stream processing on a multicore machine," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 617–629, 2017.

[49] G. Gu, R. Perdisci, J. Zhang, W. Lee, *et al.*, "Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.," in *USENIX Security Symposium*, vol. 5, pp. 139–154, 2008.

[50] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–8, IEEE, 2014.

[51] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks.," in *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, vol. 7, pp. 249–264, 2010.

[52] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation.," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 67–82, 2017.

[53] P. Jesus, C. Baquero, and P. S. Almeida, "A survey of distributed data aggregation algorithms," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 381–404, 2014.

[54] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[55] Google, "Google data centers efficiency." `https://www.google.com/about/datacenters/efficiency/`, February 2018.

[56] D. Kliazovich, P. Bouvry, and S. U. Khan, "Dens: data center energy-efficient network-aware scheduling," *Cluster computing*, vol. 16, no. 1, pp. 65–75, 2013.

[57] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 338–347, ACM, 2010.

[58] sFlow, "sflow sampling rates." `http://blog.sflow.com/2009/06/sampling-rates.html`, June 2009.

[59] D. Zobel, "Does my cisco device support netflow export?." `https://kb.paessler.com/en/topic/5333-does-my-cisco-device-router-switch-support-netflow-export`, June 2010.

[60] Wikipedia contributors, "Netflow — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=NetFlow&oldid=823922835`, 2018. [Online; accessed 23-February-2018].

[61] N. Fevrier, "Netflow, sampling-interval and the mythical internet packet size." `https://xrdocs.github.io/cloud-scale-networking/tutorials/2018-02-19-netflow-sampling-interval-and-the-mythical-internet-packet-size/`, February 2018.

[62] Cavium, "Cavium / xpliant cnx880xx product brief." `https://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf?x=2`, 2015.

[63] Netronome, "Opennfp." `https://open-nfp.org/`, 2018.

[64] N. Bjorner, M. Canini, and N. Sultana, "Report on networking and programming languages 2017," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 5, pp. 39–41, 2017.

[65] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*, pp. 684–695, Springer, 2006.

[66] Redis, "Redis." https://redis.io/, February 2018.

[67] Caida, "The caida anonymized internet traces 2015 dataset." http://www.caida.org/data/passive/passive_2015_dataset.xml, 2015.

[68] B. Wheeler, "A new era of network processing," *The Linley Group, Technical Report*, 2013.

[69] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87–95, July 2014.

[70] L. Rizzo and M. Landi, "Netmap: Memory Mapped Access to Network Devices," in *Proc. ACM SIGCOMM*, 2011.

[71] J. Preshing, "This hash table is faster than a judy array." http://preshing.com/20130107/this-hash-table-is-faster-than-a-judy-array/, January 2013.

[72] Intel, "Intel® sse4 programming reference," 2007.

[73] G. Kumar, A. Narayan, and P. Gao, "Yet another packet simulator." https://github.com/NetSys/simulator.

[74] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "phost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, p. 1, ACM, 2015.

[75] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 435–446, ACM, 2013.

[76] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 121–136, ACM, 2017.

[77] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016.

[78] EdgeCore, "Wedge 100 data sheet." `https://www.edge-core.com/_upload/images/Wedge_100-32X_DS_R04_20170615.pdf`, June 2017.

[79] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[80] N. Williams, S. Zander, and G. Armitage, "A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 5, pp. 5–16, 2006.

[81] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using machine learning technliques to identify botnet traffic," in *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pp. 967–974, IEEE, 2006.

[82] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks.," in *NSDI*, vol. 10, pp. 249–264, 2010.

[83] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 123–134, ACM, 2013.

[84] H. E. Egilmez, S. Civanlar, and A. M. Tekalp, "An optimization framework for qos-enabled adaptive video streaming over openflow networks," *IEEE Transactions on Multimedia*, vol. 15, no. 3, pp. 710–715, 2013.

[85] M. Hicks, J. T. Moore, D. Wetherall, and S. Nettles, "Experiences with capsule-based active networking," in *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings*, pp. 16–24, IEEE, 2002.

[86] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), pp. 11:1–11:16, ACM, 2018.

[87] J. C. Beard, P. Li, and R. D. Chamberlain, "Raftlib: a c++ template library for high performance stream parallel processing," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 96–105, ACM, 2015.

[88] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, *et al.*, "drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 1–14, ACM, 2017.

[89] B. Wheeler, "A new era of network processing," *The Linley Group, Tech. Rep*, 2013.

[90] A. Seznec, "A case for two-way skewed-associative caches," *ACM SIGARCH computer architecture news*, vol. 21, no. 2, pp. 169–178, 1993.

[91] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, ACM, 2014.

[92] A. Botta, A. Dainotti, and A. Pescapé, "Do you trust your software-based traffic generator?," *IEEE Communications Magazine*, vol. 48, no. 9, 2010.

[93] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching.," in *NSDI*, pp. 407–420, 2017.

[94] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[95] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 3–14, ACM, 2014.

[96] D. Shan, W. Jiang, and F. Ren, "Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 118–126, IEEE, 2015.

[97] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pp. 73–82, ACM, 2009.

[98] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, (New York, NY, USA), pp. 78–85, ACM, 2017.

[99] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 479–491, ACM, 2015.

[100] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks.," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, vol. 14, pp. 71–85, 2014.

[101] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 615–626, ACM, 2010.

[102] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks.," in *NSDI*, pp. 719–733, 2017.

[103] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, Aug 2000.

[104] CAIDA, "Statistics for caida 2015 chicago direction b traces." `https://www.caida.org/data/passive/trace_stats/`, 2015.

[105] CAIDA, "Trace statistics for caida passive oc48 and oc192 traces – 2015-12-17." `https://www.caida.org/data/passive/trace_stats/`, December 2015.

[106] O. Michel, J. Sonchack, E. Keller, and J. M. Smith, "Packet-level analytics in software without compromises," in *HotCloud*, 2018.

[107] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 3, p. 12, 2016.

[108] P. R. Jelenković and X. Kang, "Characterizing the miss sequence of the lru cache," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, pp. 119–121, 2008.

[109] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *2012 24th International Teletraffic Congress (ITC 24)*, pp. 1–8, IEEE, 2012.

[110] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: why it matters and how to model

it," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 5, pp. 5–12, 2013.

[111] A. Feldmann and W. Whitt, "Fitting mixtures of exponentials to long-tail distributions to analyze network performance models," *Performance evaluation*, vol. 31, no. 3-4, pp. 245–279, 1998.

[112] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[113] U. Dhawan and A. DeHon, "Area-efficient near-associative memories on fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 4, p. 30, 2015.

[114] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger, "Network monitoring as a streaming analytics problem," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 106–112, ACM, 2016.

[115] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 175–180, ACM, 2014.

[116] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "vcrib: Virtualized rule management in the cloud.," in *HotCloud*, 2012.

[117] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proc. SIGCOMM*, 2011.

[118] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang, "Serverswitch: A programmable and high performance platform for

data center networks.," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, vol. 8, pp. 2–2, 2011.

[119] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Enabling practical software-defined networking security applications with ofx," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[120] A. Shieh, S. Kandula, and E. G. Sirer, "Sidecar: building programmable datacenter networks without programmable switches," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 21, ACM, 2010.

[121] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my asic!," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 25–30, ACM, 2012.

[122] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev. (CCR)*, vol. 38, no. 2, 2008.

[123] Caida, "Statistical information for the caida anonymized internet traces." `http://www.caida.org/data/passive/passive_trace_statistics.xml`, February 2018.

[124] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 15–28, ACM, 2009.

[125] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 97–108, ACM, 2013.

[126] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," USENIX, 2014.

[127] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 525–538, ACM, 2016.

[128] L. Deri and N. SpA, "nprobe: an open source netflow probe for gigabit networks," in *TERENA Networking Conference*, 2003.