UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUCAS BARBOSA CASTANHEIRA

Tracing and Troubleshooting In-Network Computation

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Alberto Schaeffer-Filho

Porto Alegre
October 2022

CIP — CATALOGING-IN-PUBLICATION

Barbosa Castanheira, Lucas

Tracing and Troubleshooting In-Network Computation / Lucas Barbosa Castanheira. – Porto Alegre: PPGC da UFRGS, 2022.

97 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Alberto Schaeffer-Filho.

1. In-Network Compute. 2. Telemetry. 3. Debugging. I. Schaeffer-Filho, Alberto. II. Título.

ABSTRACT

There is a growing move to offload functionality, e.g., TCP or key-value stores, into the network – either on SmartNICs or programmable switches. While offloading promises significant performance boosts, these programmable devices often provide little visibility into their performance. Moreover, many existing tools for analyzing and debugging performance problems, e.g., distributed tracing, do not extend into these devices.

Motivated by this lack of visibility, the first half of this work presents the design and implementation of Foxhound, an observability framework for in-network compute. This framework introduces a co-designed query language, compiler, and storage abstraction layer for expressing, capturing and analyzing distributed traces and their performance data across an infrastructure comprising servers and programmable data planes.

While Foxhound is our proof of concept for flexible in-network tracing, we discovered that the traditional tracing paradigm which Foxhound embodies can suffer from scalability issues given hardware limitations of programmable data planes. In our effort to mitigate this, we identified a subset of common tracing queries that could be hyper-optimized even beyond Foxhound's capabilities. These optimizations represent a departure from traditional tracing and constitute another framework, Mimir, presented in the latter half of this work. Mimir trades-off flexibility for efficiency by exploring a set of design choices that optimize for common diagnosis and localization tasks. Our evaluations using three representative offloaded applications on an Intel Tofino-based testbed, an emulator and a simulator show that Mimir can support a subset of common tracing tasks at scale with significant lower overheads than Foxhound. Moreover, our experiments with an in-network-compute-enhanced DeathStarBench "social network" microservice demonstrates the usefulness of our approach for end-to-end diagnosis.

Keywords: In-Network Compute. Telemetry. Debugging.

Telemetria e diagnóstico para computação *in-network*

RESUMO

Há um movimento crescente para descarregar a funcionalidade, por exemplo, TCP ou armazenamentos de valores-chave, na rede - em SmartNICs ou planos de dados programáveis. Embora o descarregamento prometa aumentos significativos de desempenho, esses dispositivos programáveis geralmente fornecem pouca visibilidade de seu desempenho. Além disso, muitas ferramentas existentes para analisar e depurar problemas de desempenho, por exemplo, rastreamento distribuído, não se estendem a esses dispositivos. Motivado por essa falta de visibilidade, a primeira metade deste trabalho apresenta o design e implementação do Foxhound, um *framework* de observabilidade para computação em rede. Esse *framework* apresenta uma linguagem de consulta, um compilador e uma camada de abstração de armazenamento coprojetados para expressar, capturar e analisar rastreamentos distribuídos e seus dados de desempenho em uma infraestrutura que inclui servidores e planos de dados programáveis.
Embora o Foxhound seja nossa prova de conceito para rastreamento flexível na rede, descobrimos que o paradigma de rastreamento tradicional que o Foxhound incorpora pode sofrer de problemas de escalabilidade devido às limitações de hardware dos planos de dados programáveis. Em nosso esforço para mitigar isso, identificamos um subconjunto de consultas de rastreamento comuns que podem ser hiper-otimizadas mesmo além das otimizações do Foxhound. Essas otimizações representam um afastamento do rastreamento tradicional e constituem outro framework, o Mimir, apresentado na segunda metade deste trabalho. O Mimir troca a flexibilidade pela eficiência, explorando um conjunto de opções de design que otimizam tarefas comuns de diagnóstico e localização. Nossas avaliações usando três aplicativos descarregados representativos em um testbed baseado em Intel Tofino, um emulador e um simulador mostram que o Mimir pode suportar um subconjunto de tarefas de rastreamento comuns em escala com *overhead* significativamente menor do que o Foxhound. Além disso, nossos experimentos com um microsserviço do DeathStarBench aprimorado por computação em rede demonstram a utilidade de nossa abordagem para diagnóstico de fim-a-fim.

Palavras-chave: Computação in-network, Telemetria, Debugging.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

INC	In-Network Compute
PDP	Programmable Data Planes
RTT	Round Trip Time
ALU	Arithmetic Logic Unit
SALU	Stateful Arithmetic Logic Unit
SRAM	Static Random Access Memory
INT	In-band Network Telemetry
RPC	Remote Procedure Call
DT	Distributed Tracing
IP	Internet Protocol
UDP	User Datagram Protocol
CPU	Central Processing Unit
ASIC	Application Specific Integrated Circuit
RAM	Random Access Memory
TCP	Transmission Control Protocol
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
NIC	Network Interface Card
DevOp	Developer-Operator
DAG	Directed Acyclic Graph
E2E	End-to-End
DSB	DeathStarBench

“Be loyal to what matters”

— ARTHUR MORGAN

ACKNOWLEDGEMENTS

To my family[1], my advisors, and more generally all of the amazing friends who accompanied me throughout this journey:

Thank you, you made this all worthwhile.

[1]Which of course includes my dogs

CONTENTS

1 INTRODUCTION

1.1 Contextualization

With the end of Moore's law (VARDI, 2014; Theis; Wong, 2017), application developers have started exploring alternative methods for accelerating applications, e.g., graphics processing units (GPUs) and tensor processing units (TPUs). Programmable data planes (PDPs) with their ability to execute arbitrary computation have recently emerged as a promising hardware accelerator. In light of this development, both industry and researchers have begun to actively investigate new design points (BENSON, 2019) for classic distributed applications by offloading functionality into the network. In this work, we refer to this design point as in-network compute functions (INCs). These new design points promise to improve performance (JIN et al., 2017; TIRMAZI et al., 2019a), scalability (SAPIO et al., 2017; LAO et al., 2021), or reliability (KANNAN; JOSHI; CHAN, 2019).

This new design point complicates traditional management and diagnosis techniques. In particular, the pre-existing network diagnosis tools, e.g., NetFlow (CISCO, 2019) or In-Network Telemetry (KIM et al., 2015), are insufficient because they focus on traditional network performance metrics and provide no visibility into the inner workings of the in-network applications. However, distributed tracing (SIGELMAN et al., 2010) which has traditionally provided visibility into applications is not supported by PDPs. Essentially, while there has been extensive work on demonstrating the viability of in-network applications (JIN et al., 2017; LAO et al., 2021; ZHU et al., 2020), designing scheduling algorithms for them (ZHENG; BENSON; HU, 2018) and developing language abstractions (SONCHACK et al., 2021), there has been relatively more minor work on developing holistic diagnosis frameworks for them.

To summarize, emerging programmable data plane frameworks lack sufficient primitives (MACE; FONSECA, 2018) to enable the rich semantic-aware diagnosis tools required by DevOps to diagnose large-scale distributed systems.

1.2 Problem Setting

Existing diagnosis frameworks either focus on diagnosing distributed applications (JAEGER, 2020), debugging stateful network elements and programs (SULTANA et al., 2017), or detecting network problems (NARAYANA et al., 2017a; GUPTA et al., 2018). However, debugging in-network compute applications (INCs) requires methods to capture, process jointly, and analyze the execution of requests across both the distributed applications running on servers and their in-network counterparts running on the programmable data planes – a relatively non-trivial task. A natural path forward is to mirror efforts to extend distributed applications into the network by extending distributed tracing unto programmable data planes. Distributed tracing has become the prevalent strategy for introspecting on networked distributed services. While some popular services offer tracing plugins, deciding what gets traced is usually a difficult trade-off between performance and expressiveness.

However, such an extension is also challenged by the programmable data plane language and hardware constraints which limit both **scalability** and **flexibility**. Optimizing systems in both of these dimensions concurrently limits our design, as there is a common trade-off between flexibility and scalability that creates a "short-blanket" situation: advancing on one end will generally make solutions recede on the other. Thus, we will deal with these problems separately in this work: The first half of this dissertation will tackle flexibility (Foxhound) and the latter half will optimize for scalability (Mimir). While the systems are substantially different, queries of both systems can interoperate and thus can be mixed-and-matched to customize the flexibility-scalability tradeoff to individual cases.

1.3 This Work

Flexible In-Network Distributed Tracing. The initial half of this work presents **Foxhound**, which aims to bridge the gap between traditional network telemetry (NARAYANA et al., 2017b; GUPTA et al., 2018; SONCHACK et al., 2018; SONCHACK et al., 2018) and distributed tracing (SIGELMAN et al., 2010) by tackling program executions which cross the boundary between distributed applications and the network. Unlike traditional network telemetry (GUPTA et al., 2018) which focuses on packet level statistics, Foxhound can capture an INC's internal program state. Specifically, Foxhound enables flexible, end-to-end diagnosis by capturing and integrating trace data across programmable data planes (switches and NICs) and x86 servers.

The core idea of Foxhound is simple: developers annotate data in their INCs, which is queried by operators at runtime. Foxhound handles everything in-between such as data collection, storage, exports and, integration. To interface with users, we introduce low-level annotations and a high-level query language. The query language allows DevOps[1] to optimize common queries with domain-specific knowledge, meanwhile, the annotations allow programmers to decompose trace analysis into local processing on switches and global processing on the controller, thereby reducing memory and bandwidth requirements.

Scalable In-Network Distributed Tracing. In the latter half of this work, we present **Mimir**, inspired by the operational approach to diagnosing large-scale distributed systems used in today's web-scale infrastructure (GUO et al., 2020a; KALDOR et al., 2017; BERG et al., 2021). Specifically, while distributed traces capture rich contextual information about the flow of execution across multiple processes (or devices), in practice, distributed traces are aggregated to determine the "expected behavior", and problems are detected by comparing traces for a new request (i.e., "observed behavior") against this aggregated notion of normalcy. Specifically, outlier detection is used to determine if a new trace is anomalous relative to the aggregated information by performing a pair-wise comparison of node characteristics or edge distribution (e.g., processing latency (SAMBASIVAN et al., 2011; HUANG; ZHU, 2021; ANAND et al., 2020) or control flow (SAMBASIVAN et al., 2011; GUO et al., 2020a) outliers).

Mimir builds on these observations to provide scalable in-network distributed tracing. *In particular, our work leverages the aggregate-based diagnosis prevalent in distributed systems to redistribute and rethink the division of labour between the computing elements and the diagnosis elements by offloading aggregate analysis to programmable devices.* Given the limitations of programmable devices, simply offloading functionality poses scalability and practical concerns. Our work addresses these concerns by exploring novel designs that decouple the storage and processing requirements of the different components of the diagnosis process and uses domain knowledge about device characteristics to place the decoupled storage and processing on different components of the devices.

In essence, while traditional distributed tracing centralizes and decouples the generation and collection of data from analysis to enable flexible tracing, Mimir argues for distributing and co-designing collection and analysis of the in-network compute aspect of tracing to enable scalability and efficiency.

[1]We use DevOp as short for Developer-Operator, by which we mean to designate workers which can work interchangeably between development and operation of a software system.

1.4 Contributions

We now break up our contributions with respect to each of our designs.
Foxhound presents the following main contributions:

- We introduce an *in-network storage layer* which allows users to either completely aggregate trace information into in-network data structures, or efficiently export this to the CPU interface present in Tofino switches.
- We design a *tracing query language and compiler* to allow users to specify customizable, high-level tracing and processing tasks that guide the instrumentation and optimization of the storage layer.
- Finally, using a popular INC (NetCache (JIN et al., 2017)), we evaluate Foxhound against several diagnosis techniques and different failure modes, highlighting the importance of query-specific optimizations through micro-benchmarks.

Mimir's main contributions are:

- We identify common design patterns for INCs and characterize the implication of these design patterns for debugging distributed applications.
- We introduce scalable design choices that allows us to efficiently extend distributed tracing into the network by rethinking the division of responsibility and appropriately mapping storage and processing to different components of a programmable switch.
- We present a prototype of Mimir, port three representative INCs to Mimir, and evaluate Mimir with a combination of a hardware testbed, a simulator, and an emulator.

1.5 Roadmap

The rest of this document is structured as follows:

- Chapter 2 presents our background on several related topics related to both network programmability, in-network compute, and distributed tracing.
- Chapter 3 presents the design of Foxhound, the flexibility-facing front of this work. This chapter brings foxhound-specific motivation, design and architecture choices, an evaluation, and finally, discussions and limitations.

- Chapter 4 presents the design of Mimir, the scalability-facing front of this work. Its substructure is the same as in Chapter 3.
- Chapter 5 presents related works and compares both Mimir and Foxhound to other works which occupy different points in the design space, discussing interesting differences and similarities whenever possible.
- Chapter 6 presents our conclusion: we summarize our technical contributions as well as the central lessons learned from this work; Finally, we discuss important threads of future work.

2 BACKGROUND

We introduce programmable data planes (Section 2.1), PDP telemetry (Section 2.2), in-network computation (Section 2.3), and finally distributed tracing (Section 2.4).

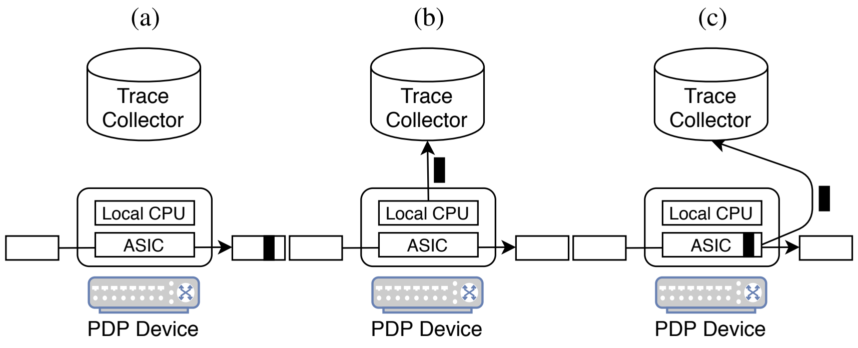
2.1 Programmable Data Planes

Programs for programmable data planes (PDPs) are largely written in the P4 language (BOSSHART et al., 2014) which is built around the match-action table abstraction. Tables perform actions associated with a set of conditions that the packet satisfies. Each match-action table is in essence a list of if-else statements that look only to the packet headers, with the only kind of persistent state being kept in SRAM registers and control-plane-driven tables that the program flow can interact with. We will now briefly discuss the language limitations and hardware limitations that the PDP architecture places on network programs.

Unlike other more traditional languages, P4 does not allow loops, floating point arithmetic, or pointer-based indirection. Additionally, current programmable data plane hardware (INTEL, 2022) is designed with limited SRAM memory and provides a limit on the data access (DANG et al., 2017). For example, during one pipeline execution, each register can only be accessed once while processing a packet and the programmable pipeline only has a fixed amount of stages and resources usable by each stage. Additionally, pipeline stages, registers, and SRAM need to be shared by all of the modules of a P4 program, compilation of a P4 program will fail if any of the resources exceed hardware limits (DANG et al., 2017). This means that telemetry should be optimized and effective, so as to justify its resource consumption. Attempts to bypass the data-access limitation or stage limitation generally revolve around packet recirculation (*e.g.*, running the packet through the pipeline again to be able to do more processing or accessing registers twice). However, recirculation comes with its own set of problems, such as recirculated packets limiting overall bandwidth (SONCHACK et al., 2021), possibly pressurizing queues and creating congestion or packet drops. For this reason, the use of recirculation is generally limited.

Finally, we mention that while PDPs offer several limitations, there have also been improvements over the state of the art of network switches. Of specific interest to this work is the more powerful PCI-based CPU port that allows the ASIC and the switch

Figure 2.1: PDP Telemetry Design Points



Source: The Authors (2022)

CPU to send packets to one another. This CPU port has more throughput than those found in previous fixed-function switches (SONCHACK et al., 2018) and can accommodate a wide range of telemetry tasks. We note, however, that the CPU port throughput (at most 32 Gbps) is still not powerful enough to sample 100% of the aggregated traffic of the programmable switch (e.g., 3.2 or 6.4 Tbps). This limitation as well as the other limitations mentioned in this section severely impact the design space of telemetry solutions.

2.2 PDP Telemetry

Today, there are three key design points for scalable network telemetry: (INT-style) appending data to the packets, (retrospection) storing data locally for analysis, (mirror) exporting packet headers to a third-party entity for analysis. Below we highlight the limitations of each design point.

- **INT-style (Fig. 2.1a):** Perhaps the most famous is to append the required data to the packet (KIM et al., 2015; BASAT et al., 2020a; JEYAKUMAR et al., 2014a; HANDIGOL et al., 2014). Unfortunately, existing techniques are extremely limited in the calculations that can be performed on the device to determine which data can be exported and also limited in the post-processing that can be done to limit data export. Furthermore, while existing techniques vary in flexibility concerning the amount of data exported and the conditions for exporting such data, when used for proactive diagnosis they export a significant amount of data. For example, traditional INT only allows exporting program states and variables without restricting export, whereas TPP (JEYAKUMAR et al., 2014a) allows for some restrictions. Both are still limited to simple program state or switch state.

- **Packet Mirror (Fig. 2.1b):** An interesting set of techniques (GUPTA et al., 2018; YU et al., 2019; LI et al., 2019) mirror packet headers to a central location for diagnosis. The central location, in turn, uses packet headers to reconstruct switch state and protocol behavior. Unfortunately, as others (KANNAN et al., 2021) have observed, such works capture only the packet headers and are generally unable to capture switch-internal runtime state where valuable information for INC debugging can be found (e.g., register state observed, table rules hit, match-action units used, hash functions applied).

- **Retrospection (Fig. 2.1c):** At the opposite end of the spectrum, recent work (KANNAN et al., 2021; WANG et al., 2020b) has demonstrated that programmable switches can store, on switch SRAM, up to approximately $O(10)$ seconds worth of diagnostic data, a sufficient amount for most network diagnostic tasks.

  However, storing observability data significantly increases the storage requirements since, by default, aggregation is not as easy to perform on request data as it is on packet data. Network telemetry queries are accustomed to working with aggregated per-flow statistics, abstracting information about individual packets. This is not trivially done for compute-centric queries, which may depend on data about individual requests. Largely due to this reason, our measurement shows that while network telemetry tasks can store up to $O(10)$ seconds worth of observability data (with per-flow aggregation (KANNAN et al., 2021)), we can only store $O(1)$ millisecond of unaggregated, per-request data (under a 10MPPS workload using a 1MB SRAM buffer to store only timestamps and identifiers) — a significant decrease in the capacity of retrospection.

2.3 In-Network Compute Functions

Programmable Networks offer the possibility of deploying the switch code as a soft solution (i.e., not etched in silicon). This allows vendors of programmable switches to sell blank programmable hardware and let users customize the behavior of this hardware.

In-network compute functions (INCs[1]) are network programs to perform computation inside the network, generally to offload applications running on x86 servers.

INCs allows for a new dimension of optimizations that goes beyond simply adding more raw hardware horsepower to x86 applications. Essentially, scalability in the real world was many times a problem which operators and system designers could buy themselves out of by adding more powerful CPUs and CPU cores. With the end of Moore's law (Theis; Wong, 2017), the need arose to explore alternative methods for accelerating applications. Offloading computation to INCs has recently (and timely) emerged as a promising post-Moore alternative to improve performance. A vast range of problems have been shown to be offloadable into the network, we now list the highly recurring ones below, as well as describe some of the underlying hardware mechanisms that make PDPs advantageous in solving these problems.

- *Caching*: This category uses the in-switch memory of programmable switches to store key-value pairs (JIN et al., 2017; YU et al., 2020; JIN et al., 2018). A common technique is to use the top-of-rack switch as a cache and leverage its advantageous position in order to intercept lookup requests and answer them even before they reach the x86 servers to which they were destined. A significant limitation of this class of work is the memory limitation imposed by programmable switches. However, it has been shown that an in-network cache can bring significant improvements to highly skewed workloads – where a few keys are highly popular and account for most of the requests. This workload distribution allows systems such as NetCache to cache these highly popular keys and respond to them at line rate, completely bypassing the server and therefore avoiding the x86 lookup cost.

- *Data Aggregation*: This class of techniques performs in-network data aggregation for higher scalability (SAPIO et al., 2017; LAO et al., 2021); Essentially, systems such as DAIET (SAPIO et al., 2017) or ATP (LAO et al., 2021) will take advantage of highly-connected network switches in the topology and use them as a distributed network of aggregators — instead of all hosts needing to communicate their results to an aggregator entity (N-to-1, possibly a bottleneck), the hosts will communicate their results to intermediary switches which will aggregate results and forward only the aggregates. This technique can be effective in increasing throughput of impor-

[1]"In-Network Computation" (INC) is a broad paradigm which has recently gained momentum along with programmable data planes. In this work we denote the network programs which carry out computation inside the network as **I**n-**N**etwork **C**ompute functions, and we use the same acronym – INC – for brevity.

tant and commonly run jobs such as distributed neural network training (LAO et al., 2021).

- *Scheduling:* Several research efforts have leveraged programmable switches to schedule requests between servers (TIRMAZI et al., 2019b; ZHU et al., 2020; LI et al., 2020). These works build upon the concept of load balancing and will consider important server-level aspects such as CPU usage to make informed decisions about how to schedule incoming requests. The advantage of PDPs is to be able to do this at line rate for billions of packets per second and to be able to use arbitrary information in order to load balance (as opposed to network-level middlebox load balancers, which focus on network-level information).

Other less prevalent problems which have been shown to be feasibly offloadable are *Machine Learning* (SANVITO; SIRACUSANO; BIFULCO, 2018; XIONG; ZILBERMAN, 2019) and *Pattern Matching* (JEPSEN et al., 2019).

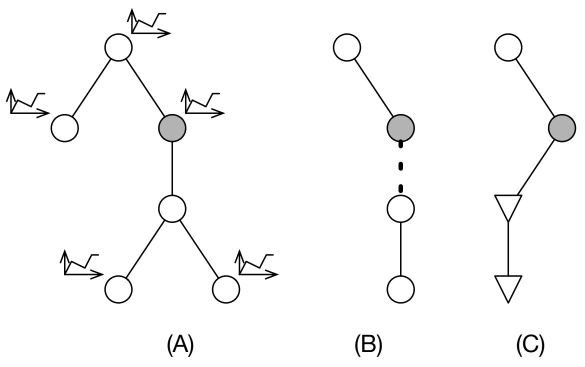
2.4 Distributed Tracing

Distributed tracers aggregate information across different hosts involved in the processing of a single request. To enable distributed tracing, developers must instrument, with annotations, their programs to store relevant context information (for example, in user-defined “baggages” (MACE; FONSECA, 2018)). At run time, the instrumentation captures context data, in atoms called “spans”, and exports to a centralized collector. The collector merges spans that belong to the same TraceID and finally reconstructs a trace, which is then stored in a database or filesystem. Essentially, the reconstructed trace is a directed acyclic graph, or DAG, that contains the spans generated in response to processing a request, illustrating the flow of control across different processes. Postprocessing happens by querying the database for existing traces and processing the query results.

Below we discuss a list of common tracing use-cases. Abstractly, traces are either analyzed in isolation or in conjunction with a group of traces. When a trace is analyzed in isolation the key objective is to determine if the trace includes well-known “anti-patterns”[2]. The more interesting scenario occurs when a trace is compared against a group. Here the goal is to either detect anomalous behavior, or to understand the collective resource utilization of the group. Below we elaborate on the most common diagnosis

[2]Patterns that occur in high correlation with undesirable outcomes.

Figure 2.2: D.T. Use Cases



Source: The Authors (2022)

use cases (KALDOR et al., 2017; SAMBASIVAN et al., 2016).

Distributed Profiling. To detect slow nodes or functions, traces with unusually long delays are generally worth investigating. To do this, traces are aggregated into a call graph (gray node Figure 2.2(a)) and annotated with the distribution of latency for the different calls. Problems are detected when a call in a new trace (Figure 2.2(b)) includes latency at the tail of the distributions.

Anomaly Detection. A request may be routed through different applications based on runtime decisions (e.g., request type or data access patterns), and identifying unusual workflows is necessary. This is similar to distributed profiling, however, the analysis focuses on the flow of execution itself. Here an anomaly is a trace (Figure 2.2(c)) that includes an unusual call (triangle node in Figure 2.2(c)): either a call that does not exist in the trace graph or a call that appears in the trace graph with low probability.

Resource Attribution. Some workflows may take a larger toll on the network than others, and we may need to determine resources used by a collection of workflows. We need to analyze a group of traces (Figure 2.2(a)) belonging to a tenant or an application to determine the resources used by either the tenant or the application.

3 FLEXIBLE IN-NETWORK DISTRIBUTED TRACING — FOXHOUND

Foxhound is our framework for flexibly implementing server-grade observability queries in the data plane. We begin this chapter by presenting the motivating insights behind Foxhound. Essentially, we wish to show the reader that INCs can fail just as x86 applications and that – unlike x86 applications – diagnosing INC failures is still a vastly unexplored challenge.

Afterward, we will elaborate on the two main components which Foxhound introduces to tackle this challenge, namely (i) our optimized storage layer – which seeks to efficiently record and expose switch-internal data – and (ii), our query language – oriented towards flexibility while also allowing users to optimize their queries and the storage layer. We finalize with our evaluation of Foxhound and important discussions.

3.1 Motivation

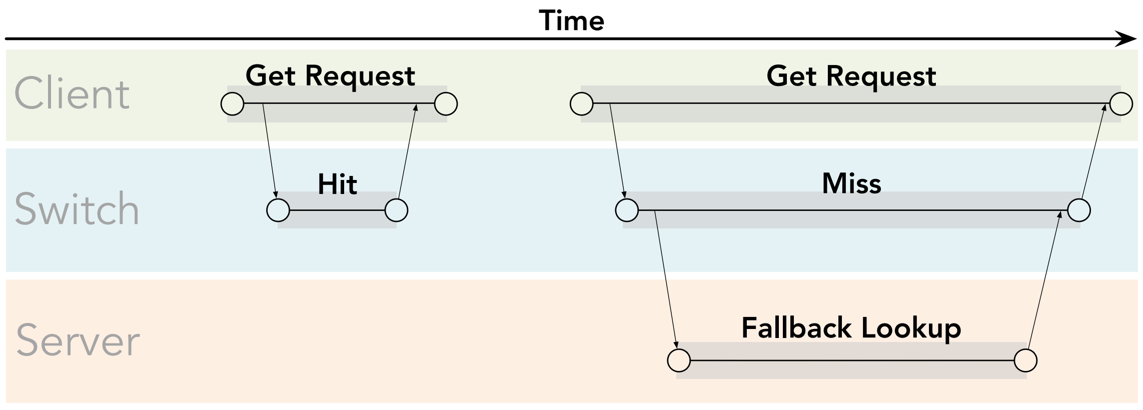
We motivate Foxhound by presenting a case study of specific INC failures, how they could be tackled in a similar vein by using traditional x86 observability, and how traditional network telemetry fails to tackle these problems.

3.1.1 INC Case Study – NetCache

To motivate the need for tracing, we describe a case study around the NetCache INC (JIN et al., 2017). In NetCache, key-value pairs are stored at ToR switches. These ToR switches identify the most frequent keys being queried and cache them. When NetCache sees a get request, the key can either "hit" the cache, in which case the switch itself will answer the request and the packet will be returned to the client with the relevant key-value, or "miss" the cache. For misses, two events happen, (i) NetCache does a fallback lookup, sending the packet to an x86 server that has the key and will reply to the request instead, and (ii) if the missing key is identified as hot, NetCache sends a cache request to its control program (running locally at the switch's x86 CPU). The control program then updates NetCache, caching the detected key.

Consider a DevOp who wishes to introspect on a storage microservice which is being accelerated by NetCache (Fig. 4.1). When NetCache is deployed, a significant

Figure 3.1: Gantt Diagram of NetCache Operations



Source: The Authors (2022)

part of lookups will happen in-network where there is no visibility other than the usual, network-centric telemetry. As we will show, network-centric telemetry systems cannot provide the level of visibility needed for troubleshooting NetCache and, more generally, INCs. We identify some of these situations:

Latency Problems. A common high level performance indicator is latency. Specifically for NetCache, there will be two latency profiles (Fig. 4.1), one for hits (i.e., fast, when the switch answers the query directly) and one for misses (i.e., slow, when the switch needs to perform a fallback lookup at an x86 server). Skewed latency distributions can indicate a stale cache in which the currently cached keys are not relevant, leading to lower hit-ratio and a higher need for slower fallback lookups. This kind of diagnosis is impractical in network-centric telemetry, given it requires end-to-end RPC latency information from both servers and switches.

Component Failure. If NetCache's x86 control program fails, NetCache will still be performing correctly, but its x86 counterpart will not. This becomes clear once we look at the DAG of a group of executions and see that some of them have missing edges responsible for updating the cache. Without application-specific information about the chains of operations happening in the devices, this error is hard to localize.

Misconfiguration. INCs such as NetCache can be misconfigured in many ways, such as the count-min sketches for identifying heavy-hitting key-value pairs being too small and producing false positives. In this case, the DAG of executions will start showing unreasonable invocation counts for an edge that was previously rare. Again, network-telemetry has critical limitations for doing this kind of analyses.

Takeaway: Compute-specific problems, which are widely studied by x86 observability efforts (SAMBASIVAN et al., 2011; GUO et al., 2020b), are still unexplored in the context of PDPs and INCs. Network-centric telemetry systems tend to ignore application-specific data and semantics of INCs, which are fundamental for problem localization and diagnosis.

3.1.2 The Many Forms of Tracing Data

Recent publications (SAMBASIVAN et al., 2016) and presentations at industry conferences (YAKIMOV, 2019; SHKURO, 2019) have shed some light on how distributed tracing data is currently being used in the wild at large scale production infrastructure. Below we discuss a list of the more common formats and use-cases (KALDOR et al., 2017; SAMBASIVAN et al., 2016).
Aggregate-based: With aggregation, one can reason about several traces at once (ANAND et al., 2020; SAMBASIVAN et al., 2011), differently from the classical "single-trace view". Aggregates generally can be used as indicators of normalcy (*e.g.*, hit-ratio of a cache or latency distribution of a lookup).
Span-Based: Traces do not need to be span-complete to be useful. Sometimes upfront sampling decisions could leave important spans untraced. In identifying a run-time condition of interest (*e.g.*, operation with latency above the 99th percentile), capturing even a single span can be useful, for example, to identify the source and destination addresses involved and try to isolate the offending component.
E2E-based: Classic distributed traces (SIGELMAN et al., 2010; JAEGER, 2020) which capture the entire “life” of a request. E2E traces make no compromise on data-richness and coverage (beyond sampling), which makes them the most heavyweight, but also the most versatile approach. Through post-processing, E2E traces can be useful for all other tasks, such as aggregation, anomalous trace detection, and resource attribution.

3.1.3 Limitations of Current Telemetry

In-network compute creates a need for compute-centric observability inside the network (BENSON, 2019). Developer-Operators (DevOps) today need cross-layer visibility (*e.g.*, Section 3.1.2) to diagnose distributed architectures. However, most of the compute-specific aspects of INCs are ignored by current network-centric telemetry systems. Furthermore, INCs are always deployed in the context of a distributed system comprised of x86 servers and network switches. Thus, we have a need to localize errors within the infrastructure, which makes device-centric debugging and switch-local telemetry insufficient. While several telemetry solutions were designed for programmable networks (NARAYANA et al., 2017a; GUPTA et al., 2018; SONCHACK et al., 2018; SONCHACK et al., 2018; SULTANA et al., 2017; KIM et al., 2015; WANG et al., 2020a; WANG et al., 2020b), the end-to-end aspect for analyzing computation is still missing. The main workflow of network telemetry remains unchanged: *to analyze what comes in and what goes out of each switch.* In-network compute forces us to shift perspective when introspecting on the network: *we must correlate events happening in-between packets in and out, throughout the infrastructure*.

Current Shortcomings: Recent works have presented several different ways of introspecting on programmable switches. However, none of these efforts efficiently combines the diagnosis classes (Section 3.1.2) necessary to holistically debug a network armed with in-network compute. These works either *(i)* are not able to match traces from switches to their respective server executions (NARAYANA et al., 2017a; GUPTA et al., 2018), *(ii)* will only export aggregated records, generally optimized to network-centric measurements and data structures (SONCHACK et al., 2018; WANG et al., 2020b). Conversely, some works will disallow aggregates and optimized data structures, only operate with heavy measurements, creating high overheads (SULTANA et al., 2017; KIM et al., 2015).

General issues with the more network-centric works include inability to reach into internal states from switch variables. Essentially, they only look at the packet *ins* and *outs*, while neglecting the computation *in-between* (NARAYANA et al., 2017a; GUPTA et al., 2018; SONCHACK et al., 2018). Finally, most works will not instrument arbitrary source codes (incurring in an error-prone manual merging of INC code and observability code).

3.1.4 Challenges of Extending Traditional Distributed Tracing

There are a few common ways to export data from switches. The fundamental challenge lies in the restrictions placed by the PDP-ecosystem as it is physically limited in storage resources and practically limited in the amount of telemetry data we can generate. Techniques such as INT (KIM et al., 2015) can achieve high throughput for exporting tracing data, however, they utilize a high percentage of the total bandwidth and will decrease the INC's performance. Foxhound argues for using switch memory as temporary storage, and optimizes the export of traces through the CPU link of switches.

Programmable switches have their x86 CPUs (control plane) interconnected to their switching ASIC (data plane). This interconnect is given through a PCI interface, which offers a low-throughput link that can be used to not significantly impact switch performance. Specifically, this link can be used to export telemetry data directly out of the data plane and into the switch's x86 CPU which avoids overusing high throughput links of switches, possibly over many hops (SONCHACK et al., 2018).

3.2 Foxhound Design

We introduce the workflow of Foxhound (Section 3.2.1).

3.2.1 Workflow

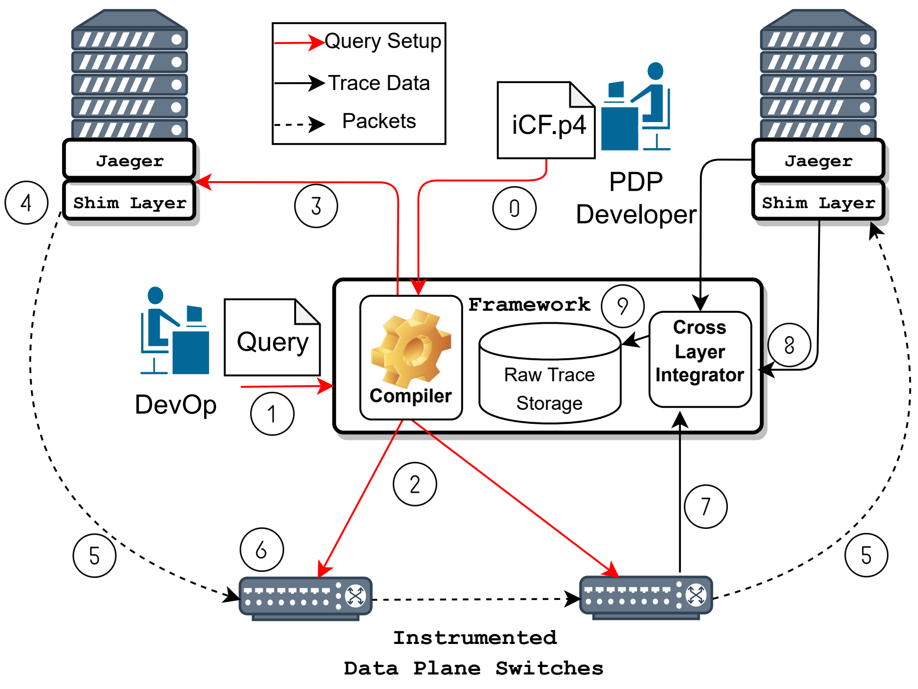
We explain our workflow with an end-to-end running example (and the corresponding *(steps)* in Figure 3.2). Consider a DevOp who wishes to introspect on a key-value store microservice which is being accelerated by NetCache (JIN et al., 2017).

Query Setup. Initially, INCs such as NetCache have their code annotated to indicate variables of potential interest *(0)*. Next, a DevOp will write the desired query to Foxhound *(1)*, which then generates and loads an optimized instrumentation to data plane switches based on previously annotated code *(2)*. Next, whenever a client tries to send a query packet to NetCache, a *shim layer* at the client (now made aware of the new query *(3)*), will tag outbound packets with an RPCID and create a stub span at the x86 trace.

RPC Tracing. The tagged packet then proceeds into the network, where it may go through one or more INCs *(5)*. Upon triggering an INC, the *instrumented switches* will store, at the switch ASIC, the state of annotated variables along with the RPCID of the tagged packet *(6)*.

Trace Export. Finally, the switch CPU continuously captures the data-plane resident information and exports it in the form of PDP spans to the *Foxhound framework (7)*. The stub spans previously inserted at the x86 traces will be filled out by matching RPCIDs

Figure 3.2: Foxhound Workflow



Source: The Authors (2022)

and infusing the x86 trace with information from Foxhound's PDP spans *(8)*. Finally, a holistic, cross-plane trace is made available to the DevOp via usual tracing interfaces (*e.g.*, Jaeger) *(9)*.

3.3 Foxhound Architecture

Next we provide a brief overview of the main components in Foxhound, as well as design choices we made.

3.3.1 Shim Layer

Foxhound needs shim layers to run on the application servers for *tagging* RPC invocations which should trigger PDP tracing, as well as explicitly creating stub spans on server traces, which denotes where to put information from PDP spans during the end-to-end merge.

Tags. In essence, the trace instrumentation on each switch uses the tags to deter-

Table 3.1: Annotation primitives

Annotation	Description
`@Store(Var)`	Stores `Var` locally at the switch
`@Sketch(Var, Args)`	Uses a sketch for optimized storage

Source: The Authors (2022)

mine if and how they should store data by pattern matching on the QueryID of the RPC packet. Query IDs are assigned to RPCs by the shim layer based on a set of user-queried conditions observed at runtime.

3.3.2 Instrumented Switches

Switches run the instrumentation created by the query compiler. Additionally, we use the x86 CPU present in switches to export data.

Annotations. In distributed tracing, programs are either manually (CHOW et al., 2014; KALDOR et al., 2017) or automatically (MACE; ROELKE; FONSECA, 2015) annotated. In INC-networks, we anticipate that INC code will be manually annotated. Fortunately, today's PDP programs are manually annotated for verification reasons (LIU et al., 2018; FREIRE et al., 2018) and we believe we can piggyback on these efforts: we argue that in addition to creating verification-oriented annotations, the developers should add annotations for tracing.

Foxhound's annotations (Table 4.3) provide hints as to how the data should be stored. Specifically, data can be stored *as-is* using the `@Store` annotation, which directly correlates to an RPCID, useful for single span and end-to-end diagnoses (Section 3.1.2); or summarized and aggregated within specialized data structures—`@Sketch`, useful for aggregate-based diagnosis, where individual RPCID information is not needed and is lost during aggregation. Annotated examples can be found later in Section 3.6.

Storage Layer. Our storage layer is responsible for (i) Efficiently storing data for which we provide two primitives, the TraceStore and TraceStats, which build upon each annotation. (2) Efficiently addressing hardware limitations of the PCI-CPU interface is physically limited and can only export a subset (max 32Gbps) of full line rate (*e.g.*, 3.2Tbps), which we do by intelligently scheduling data transfers from ASIC to CPU and effective rate limiting which data is stored. The idea is that we cannot circumvent the hardware limitations on exports without abusing in-network links, we prefer to allow users to dynamically prioritize queries and arbitrarily partition the CPU-link between the queries.

3.3.3 Foxhound Framework

Implements the query compiler that optimally configures Foxhound with information extracted from the queries. The compiler can configure sketch structures (as per user annotations, Section 3.4) and post-processing can be partitioned and pushed into PDP devices (as per user queries Section 3.5) for more efficiency. It also comprises the cross-layer integrator, where trace data obtained from the switch CPU of PDP devices is collected and merged into trace data from x86 servers.

Trace Query Language. The goal of our query language is to provide a high-level interface for capturing operator tracing policies, an interface that is expressive for the general case while also simultaneously capturing the subtle details required by Foxhound to optimize for efficiency, performance, and accuracy. Our query language builds on a well accepted tracing language for expressing tracing requirements: Baggage (MACE; FONSECA, 2018). In particular, while baggage traditionally captures the data to collect and causally correlate, we extend it to also capture the operator's intended processing and analysis requirements (written as `procedures` which operate on the annotated variables). Describing the processing and analysis requirements is the lynch-pin in minimizing storage overheads as this allows us to either restrict what we store (using early filtering primitives at the ASIC) or in ideal situations to only store summaries of the collected information (using @Sketch annotations).

Query Compiler & Cross-Plane Integrator. Once the annotated source codes and DevOp queries are submitted to Foxhound, the query compiler will generate an instrumented version of the annotated P4 code to be loaded on the switches. The instrumentation includes code to capture the state of variables at their annotated points in the code, as well as domain-specific query optimizations, and the instantiation of our storage layer. Finally, after the instrumented source code is loaded, query setup is complete and control passes to the query engine. The main objective of the cross-plane integrator is merging PDP traces into server traces, finalizing post-processing, and exporting the integrated traces to a traditional trace storage backend (such as the Jaeger collector) where DevOps can access the cross-plane traces.

3.4 Storage Layer

We discuss the intuition behind our storage layer (Section 3.4.1), detail the layer's abstractions (Section 3.4.2), and describe its scheduling algorithms (Section 3.4.3).

3.4.1 Storage Intuition

Intuitively, the different use-cases for tracing (§ 3.1.2) require different type of data. Specifically, aggregate-based tracing enables aggregated data (SONCHACK et al., 2018; GUPTA et al., 2018; NARAYANA et al., 2017a) while span-based and E2E-based require raw switch state (SULTANA et al., 2017). These two approaches explore different points in the design of diagnostic systems with the former enabling scalable diagnosis but sacrificing precision, while the latter enables precise packet level diagnosis, it suffers from scalability limitations. Foxhound aims to support both extremes while providing the developers with a query language to control these trade-offs. By explicitly designing for trace-specific usecases, Foxhound improves storage efficiency and by tailoring storage, at runtime, to queries, Foxhound maximizes storage use.

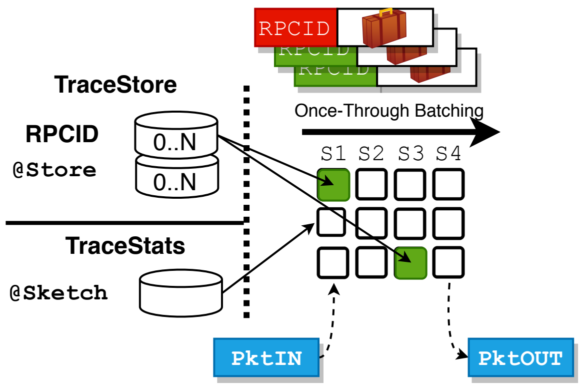
3.4.2 Storage Abstractions

Our storage layer exposes two different primitives:

TraceStore. TraceStore maintains raw trace data and acts as an append-only log for storing spans from tagged invocations (which are the RPCID from the tag, and the state from relevant annotations). Traditionally, storing raw data is unscalable. However, the sole goal of the TraceStore is not to store an arbitrarily large amount of trace data but to capture sufficient trace data to enable optimizations that batch and transfers the data to the CPU.

Implementation: Programmable switches provide two main storage hardware: tables and registers. Tables are unfortunately too slow to update at line rate, and thus we design TraceStats atop the register array hardware structure. We implement Foxhound's TraceStore using a circular buffer. The logical representation of TraceStore is shown in Fig. 3.3. Specifically, we instantiate an N-slot circular buffer for each annotation and one for storing RPCIDs. However, we will also want to replicate these register arrays

Figure 3.3: Pipeline Piggybacking.



Source: The Authors (2022)

throughout the pipeline stages in order to be able to batch spans (explained further into the section).

TraceStats. TraceStats consist of a set of sketch data structures for effectively capturing device-local histograms and distributions. These structures are instantiated on demand according to Sketch annotations within the code.

Implementation: Significant work exists on instantiating sketches on PDPs (LIU et al., 2016; ZHANG et al., 2021; LIU et al., 2019b). Foxhound's TraceStats builds on this rich literature of work. While prior work generally focus on create general sketches, Foxhound tailors and codesigns TraceStat's sketches to the specific type of data being captured and the query primitive being applied on the data. In Table 3.2, we provide a non-exhaustive list of these structures and highlight example procedures that they map to. The hardware structures, e.g., sketches, are co-designed with the query language's procedures, with the access methods for each structure being defined on the procedures. One could extend the structures of Foxhound by implementing a hardware sketch (in P4) and a procedure with the necessary access methodology. In short, we support hardware structures that effectively *aggregate* data required for coarse grained analyses (overall network and INC health indicators).

3.4.3 Scheduling and Rate Limiting Algorithms

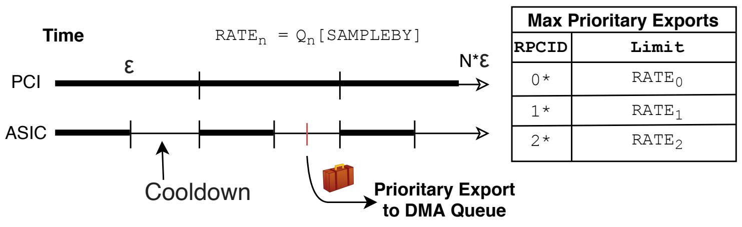
Foxhound-instrumented switches need to have a pipeline scheduling algorithm to coordinate the export strategy as well as rate limit queries.

Table 3.2: TraceStats Sketches

Sketch	Example Procedure	Input Args	Memory
`Array`	`ControlDstr.`	`Width`	`Width`
`Histogram`	`LatencyDstr.`	`BinSz,Max`	`Max*BinSz`$^{-1}$
`Matrix`	`Attribution`	`#Rows,#Cols`	`#Rows*#Cols`
`EWMA`	`MovingAverage`	n	n

Source: The Authors (2022)

Figure 3.4: Rate Limiting.



Source: The Authors (2022)

PCI Scheduling. Naively, exporting individual register arrays results in significant under utilization of an already limited resource (i.e., the switch's CPU-link (SONCHACK et al., 2018)). A more common approach is to have the switch generate a packet which is then recirculated through the ASIC multiple times – each time the packet recirculates multiple arrays are appended to the packet. This approach improves the throughput of the switch CPU-link however, the act of recirculating reduces the throughput for traffic being forwarded by the switch.

Motivated by these short comings, Foxhound's explore a different point in the design space inspired by StarFlow (SONCHACK et al., 2018). The idea is to carefully select the stateful registers used to instantiate the TraceStore within the switch's pipeline in a way that spans can be batched in a "once-through" fashion (Fig. 3.3). Whenever a packet is received, concurrently to that packet's original processing path, Foxhound will read spans from the buffer area and batch them. These batches of spans will piggyback on the ordinary packets up until the end of the pipeline, where the batched spans will be cloned, the batch is sent to the CPU and the original packet is transparently forwarded.

PCI Rate Limiting. Exporting data with the PCI takes much more time than it takes for the TraeStore's log to fill up. To ensure that Foxhound only attempts to export data when the PCI is available for exporting, we introduce a cooldown period (Figure 3.4) which ensures that there exists a minimal time gap of length d between two PCI exports. Gaps smaller than d are forcefully dropped. By only allowing exports with gap greater than d, we (i) keep the CPU link close to its actual limit and do not overuse the packet replication engine, (ii) keep the queue of the CPU-link relatively free. While the former allows us to be more efficient, the latter powers our mechanism for rate-limiting and prioritizing queries and attempting to fulfill sampling guarantees.

To enable this "cooldown" period, we leverage a unique and rarely used hardware resource: the queue of the CPU port. Essentially, we rate limit packets by storing packets in the CPU port queue according to priority.

Algorithm 1 PIPELINESCHEDULING(PKTIN)

```
 1: if not OnCooldown() or PriorityExport(PktIN.Query) then
 2:     Export=True //Free to Export
 3: else
 4:     Export=False // On Cooldown, save span to memory
 5: end if
 6: if not Export then
 7:     RegPtr ← (RegPtr+1)%BatchSize
 8:     Regs[RegPtr][WriteIDX] ← PktIN.span
 9:     if RegPtr == 0 then
10:         WriteIDX ← (WriteIDX + 1) % BufferSize //Wrote 1 complete batch
11:     end if
12: end if
13: if Export then
14:     ExportHdrs ← ExportHdrs + PktIN.span
15:     for i=0; i < BatchSize; i++ do
16:         ExportHdrs ← ExportHdrs + Reg[i][WriteIDX - 1] // Get last full batch
17:     end for
18:     LastReg[0] ← LastReg[0]-(BatchSize+1)
19:     CloneToCPU()
20: end if
```

Implementation Details: Algorithm 1 showcases our pipeline scheduling strategy for the data plane. The algorithm brings two main concepts, batching and rate limiting the exported traces in order to enforce Foxhound's prioritized sampling rates. In Line 1, Cooldown is discovered by saving the timestamp of the last export and comparing to the current timestamp to see if a minimum amount of time has passed. Whenever exports are allowed, Foxhound will opportunistically piggyback a single batch of spans together with the packet being exported (Line 13–20). This is done by concatenating the current packet's span (Line 14) with previously recorded spans from the other buffer instances (lines 15–17). Contrarily, if the storage layer is on cooldown (Line 4), Foxhound will commit the span to memory. Lines 6 to 11 do basic bookkeeping to store spans among N (the predefined batch size) circular buffers.

3.5 Query Language

In this section, we present an overview of our query language (Section 3.5.1), a detailed description of its constructs (Section 3.5.2), and our strategy for partitioning the trace computation within the infrastructure (Section 3.5.3).

3.5.1 Query Language Overview

Our aim in designing our query language is to make it simple to express diagnosis tasks traditionally applied to distributed tracing while hiding crucial hardware details. These diagnosis tasks generally focus on collecting **data**, *e.g.,* latency or error conditions, for a **specific subset** of the infrastructure, *e.g.,* applications or tenants, under certain **conditions**, *e.g.,* high load. To extract insights, diagnosis data is **post-processed**. Lastly, to scale, certain queries are **sampled**.

In designing our query language, we take inspiration from SQL – a query language currently used by DevOps to analyze existing diagnosis data[1]. We note that, unlike other PDP query languages, the body of queries will specify DevOps-relevant parameters, with the low-level processing needs defined on postprocesses which DevOps can invoke as black boxes. The grammar for our query language is described in Figure 3.5. At a high level, the language allows DevOps to select: what data they are interested in collecting; from which services this data should be collected; in which conditions should this data be collected; and how should this data be sampled. Table 3.3 summarizes the constructs.

Table 3.3: Foxhound Query Language Constructs

Construct	Description
`SELECT(What)`	Specifies postprocessing for the trace
`FROM(Selection)`	Defines to which entities the trace applies
`WHERE(Cdtn)`	Describes preconditions for tracing
`SAMPLEBY(SampleType)`	States the sampling parameters of the trace
`FILTER(Args)`	Allows early filtering of trace data

Source: The Authors (2022)

[1]Diagnostic data is often stored in Cassanda, MySQL, Redis.

Figure 3.5: Foxhound query syntax

```
⟨Query⟩ ::= SELECT ⟨What⟩ FROM ⟨Selection⟩
     WHERE ⟨Cdtn⟩ SAMPLEBY ⟨SampleRate⟩
     FILTER ⟨FilterOp⟩

⟨What⟩ ::= procedure(args)

⟨Selection⟩ ::= ⟨appID, incID, srcID, dstID⟩

⟨Cdtn⟩ ::= ⟨Cdtn⟩ ⟨Op⟩ ⟨Cdtn⟩
  | ⟨ContextOp⟩

⟨SampleRate⟩ ::=p, lowerBound

⟨FilterOp⟩ ::= (annotated_var,⟨Op⟩,value)

⟨ContextOp⟩ ::= context[key] ⟨Op⟩ value

⟨Op⟩ ::= ≤ | ≥ | == | !=
```

Source: The Authors (2022)

3.5.2 Query Language Constructs

We now describe the clauses as well as highlight important design nuances we imbued in the language.

SELECT: Much like the SQL statement, SELECT specifies the trace data to collect. This is done by either *(i)* directly specifying the low-level annotated variables to be traced and/or *(ii)* specifying high-level procedures (either built-in or user-defined) that will perform some form of post processing to the traced variables.

FROM: The goal of the FROM clause is to define the entities or predicates from which the trace data is collected. Entities are expressed as a tuple of appID, INCID, srcID, dstID. The elements of this tuple represent, respectively, a high level application (which invokes INCs), a target INC, a source and a destination. Programmers can assign wildcard values to any of the elements of the tuple.

WHERE: The WHERE statement allows DevOps to specify runtime conditions for a query. RPCs that satisfy those conditions should be tagged with the appropriate QueryID by the shim layer. In particular, the WHERE clause supports logical expressions which involve a context object. Applications can write arbitrary runtime data on the context object for queries to be able to access. For example, the application can write, at each RPC, the current USERID to context; Later, DevOps will be able to write a Foxhound query $Q1$ for a specific value of USERID. This allows DevOps, for example, to focus on

$Q1$ packets by either not sampling packets for that query or even specifically prioritizing $Q1$ packets using Foxhound's prioritization scheme.

SAMPLEBY: Sampling is instrumental in scaling distributed tracing and monitoring in general. Our SAMPLEBY construct represents a departure from traditional SQL style queries. Essentially, usual traces such as Jaeger will make an upfront decision of whether to trace a request or not based on a sampling rate. One of two jobs of our SAMPLEBY construct is to negate Jaeger's decision and, with probability p, not trace the in-network part of the invocation[2]. This is because switch throughput may not be enough to fulfill even the sampled rate mandated by Jaeger to its x86 spans. We argue that DevOps should not change x86 parameters due to PDP limitations, and this is why Foxhound leaves Jaeger's sampling untouched, SAMPLEBY is only for the data plane.

This double sampling may become too aggressive and hurt accuracy. Given that we cannot circumvent the hardware limitations on exports without abusing in-network links, we prefer to allow users to dynamically prioritize queries and arbitrarily partition the low-overhead CPU-link between the queries. Essentially, each query's *SAMPLEBY* takes a second argument $lowerBound$ that parameterizes Foxhound's mechanism for query rate-limiting (Section 3.4).

FILTER: The filter construct serves to guide Foxhound's optimization of queries by pushing processing down into the programmable ASIC[3].

PROCEDURES: Foxhound's PROCEDURES are functions which allow DevOps to control and perform post processing on the traced data. This notion of procedures is similar to "stored procedures" in SQL-based systems — in a similar vein, Foxhound provides a set of INC-independent built-in procedures (summarized in Table 3.4) and allows DevOps to create their own INC-specific procedures. The insights behind the PROCEDURE construct are many; Removing explicit, low-level processing from queries promotes a better separation: the query focuses on parameterizing trace behavior while procedures encapsulate low-level processing; Writing procedures as ordinary programs allows us to extend all of the functionality of the host language, instead of pre-defining a constrained set of processing functionalities.

[2]SAMPLEBY can also override the decision made by the WHERE clause.

[3]In our current prototype, this filtering is manual and is done with a hard threshold specified by the user. Alternatively the threshold could be dynamically changed using table rules or stateful registers

Table 3.4: Built-in Procedures

#	Procedure	Annotation	Primitive	Description
`Q1`	`SlowHosts(Percentile)`	`Latency`	Store	Discovers the sources of requests with latency > `Percentile`.
`Q2`	`HitRatioAnomaly(Low,High)`	`Hit`	Store/Sketch	Localizes situations where cache hit-ratio was outside a specific range.
`Q3`	`SpectroScope(Threshold)`	`pkt.op`	Store	Builds an RPC call graph and discovers anomalous edges using (SAMBASIVAN et al., 2011).
`Q4`	`LatencyDistribution()`	`Latency`	Store/Sketch	Aggregates latency into a histogram
`Q5`	`Latency(Threshold)`	`Latency`	Store	Traces events with latency higher than a threshold
`Q6`	`INCTrigger()`	`Flag`	Store	Filters out spans that did not trigger INCs.
`Q7`	`Attribution()`	-	Store/Sketch	Maintain per switch port, per-INC invocation counts
`Q8`	`Visited()`	-	Store/Sketch	Marks the switches reached by specified executions
`Q9`	`ErrCode()`	`ErrCode`	Store	Stores P4-specific Error Codes.
`Q10`	`ErrCodeDist()`	`ErrCode`	Store/Sketch	Calculates per-switch distribution of P4-specific Error Codes.

Source: The Authors (2022)

3.5.3 Query Partitioning

Foxhound's optimizations are focused on creating an early reduction in the tracing dataflow by opportunistically fragmenting postprocessing operations (either aggregation or filtering) and embedding of a subset of these within the data plane switches, so as to avoid the switch export bottleneck and ease CPU load. In Foxhound these optimizations are (i) in-switch span `FILTERS` and (ii) sketch primitives (in-network data aggregation). Sketches have their own fixed processing dictated by annotations. But FILTERs are query-specific.

Filtering Spans We note that partitioning procedures as to where each computation executes inside the infrastructure (i.e., switch ASIC, switch CPU, or framework) can yield effective optimizations on the overall flow of data. For example, assume a procedure that drops spans which do not meet a certain condition. To *filter* out spans directly at the ASIC means that (i) unused span will not be exported through the PCI and (ii) neither will they need to be filtered at the framework. If the condition cannot be expressed in P4, we could still place the filter on the switch CPU rather than on the framework, which helps unburdening the latter.

Figure 3.6: End-to-End Instrumentation

(a) Query

```
SELECT SlowHosts("P99")
FROM ("DSB","NetCache",*,*)
WHERE Context["Time"] < 06:30
FILTER("Latency",">", 1ms)
```

(b) Annotated Source Code

```
#DevOp Annotated NetCache.p4
if(pkt.op==WRITE)
    CacheWrite(pkt.key,pkt.val);
if(pkt.op==READ)
    if(isCached(pkt.key)) #Hit
        Hit=1;
        pkt.val=CacheRead(pkt.key);
        ipv4.dst = Sender;
    else #Miss
        Hit=0;
        hdr.NC.begin=now();
        ipv4.dst = Server;
        if(HotKey(pkt.key))
            CloneToCPU=True;
if(pkt.op==READREPLY)
    Latency=NOW()-hdr.nc.begin;
    @Store(Latency)
    @Store(ipv4.source)
@Sketch(Array,idx=Hit,+1)
@Store(pkt.op)

Forward(Ipv4.dst);
```

(c) Instrumented Source Code.

```
if(pkt.op==READREPLY)
    Latency=NOW()-hdr.nc.begin;
    #Previously: @Store(Latency)
    if(Latency > P99_REGISTER[0])
        RPCID_REG[next]=pkt.RPCID;
        LATENCY_REG[next]=Latency;
```

Source: The Authors (2022)

3.6 End-to-End Foxhound

We now present and end-to-end rundown of Foxhound's workflow and optimizations with a complete example.

Compilation and Optimization. Initially, the system is given a query (Fig. 3.6a) and annotated source code of an INC (Fig. 3.6b). Foxhound's compiler then creates instrumented source code (Fig. 3.6c) to be compiled by P4C, and initialization commands to be run at switches. The filter operation in our example will modify the instrumentation of the `@Store(Latency)` annotation[4].

Shim-Layers are informed of all queries and will decide, for each invocation, the relevant query by evaluating the `WHERE` condition. In our example, NetCache invocations from the DSB's SocialNetwork application occurring before 6:30 AM will be tagged and assigned to query zero. The absence of `SAMPLEBY` presumes 100% sampling rate.

Runtime and Cross-Plane Trace Integration. Tagged packets will be run through our pipeline scheduling algorithm Algorithm 1. The PDP spans which are succesfully exported will be sent to the framework for merging.

The cross plane integrator finds each x86 span containing RPCID references (server spans created by the shim-layers point to PDP spans through their RPCID). The integration is then reduced to de-referencing the PDP pointers and actually hard-coding PDP spans into the original server spans (PDP spans are disguised as server spans, but identified as originated from the network). Finally, we leverage Jaeger (JAEGER, 2020) for exporting the now augmented server spans and making end-to-end, cross-plane traces available to DevOps.

3.7 Prototype

Compiler: The Foxhound P4-to-P4 translator is written in 1100 lines of Python Code. We fill the templates by using Python F-strings, which allows our program to run compiler logic inline with P4 code (similarly to templates of BeauCoup (CHEN et al., 2020)). All P4 storage layer templates are written in 88 lines of P4-14 and 198 lines of Python. All of our P4-Tofino compilations used Barefoot SDEs 8.9.1 (legacy, due to the fact that NetCache was written for an old version of the SDE) and Intel P4 SDE 9.4.0. We discover Tofino pipeline resource usage by using the P4insight tool from SDE 9.4.0.

[4]For simplicity, we omit the instrumentation of the sketch annotation that was showcased in the figure

Shim Layer and Switch CPU agent: The shim layer and switch-CPU agents are written in a total of 2422 lines of Python Code. To read/write to stateful switch elements, our switch-CPU agent uses either the PCI-based API of the Tofino ASIC (for a hardware deployment) or the Thrift API of BMv2 (P4LANG, 2020) (for emulated topologies). For rate-limiting on our hardware testbed we used tc. For our E2E deployment, we instantiate the social network microservice of DSB (GAN et al., 2019) using Docker (MERKEL, 2014; DOCKER...,). In all of our deployments we use a 1 second interval between polling @Sketch annotations (1 poll per second). While this was sufficiently accurate for our evaluations, early experiments suggest that much higher frequencies are attainable using the Tofino API (up to 1M polls per second for 10-slot registers).

Query Engine and E2E Integration: For end-to-end tests, we use the Jaeger collector (JAEGER, 2020) with Badger file-system backend. We leverage Jaeger's JSON endpoint for reading x86 traces and writing back the integrated, cross-layer traces with the added Foxhound spans. For standalone Foxhound experiments, we built a customized collector.

3.8 Evaluation

Our evaluation of Foxhound is motivated by the following questions: (i) How well does Foxhound perform relative to state-of-the-art diagnosis systems? What are the overheads? (§ 3.8.3), and (ii) How do the individual components of Foxhound enhance its effectiveness and feasibility on current programmable hardware? (§ 3.8.4).

3.8.1 Experiment Setup

We evaluate Foxhound using a combination of large scale emulation to explore the benefits of Foxhound at a larger scale and a small hardware testbed to understand and microbenchmark Foxhound's performance characteristics.

Emulator setup: We use the BMv2 emulator (P4LANG, 2020) to emulate a Fat-Tree $k = 4$ topology. We evaluate 3 different queries (Table 3.4). The experiments were executed on a virtual machine running with 4 cores and 16GB of memory. The clients and server are located on randomly selected pods to ensure traffic traverses the entire topology.

Hardware Testbed Setup: Our hardware testbed comprises 2 servers with 8-core Intel Core i7-9700 CPU @ 3.00GHz, 16GB RAM, running Ubuntu 18.04 with kernel version 4.15. The servers are connected through 40Gbps Agilio LX SmartNICs to a Wedge 100BF-32X 32-port programmable switch powered by a 3.2Tbps Tofino ASIC and a quad-core Intel D1517 x86 CPU @ 1.6GHz. For NetCache experiments, we use 1 server as the client and the other as the server. Our workload uses the same distribution as the NetCache paper (JIN et al., 2017), with NetCache requests being mixed with random background traffic in a 1:1 ratio. This is important because we also want to see the effect of background traffic on approaches which do not discriminate INC packets from the background traffic (*e.g.*, INT).

Telemetry Systems. We compare Foxhound against different classes of telemetry techniques.

Inband Network Telemetry. (e.g., INT, PINT (BASAT et al., 2020b)) which, contrary to Foxhound, stores network-centric data on packet headers and extracts those headers at end-hosts for processing. We note that we refer to INT as the real-world solution (INT. . . ,) which uses network switches as INT sources to create headers which record per-packet network-centric data and then remove those headers and export them prior to packet delivery (at INT sinks). Each hop collects conventional data such as SwitchID, timestamps, and egress port. We do not refer to INT as the open specification of any in-network system which includes data in-header (which is a fundamental technique in our own work, and not a competing technique). Finally, we use statistical outlier detection to detect problems.

Jaeger. Works (JAEGER, 2020; ZIPKIN, 2021; FONSECA et al., 2007) that collect traces from x86-applications and stores them at a central collector. We instrument the applications with annotations, *e.g.*, for NetCache we annotate the client and cache server. We reuse Spectroscope (SAMBASIVAN et al., 2011) localization techniques to detect problems.

Optimized Foxhound. Whenever possible, we will show alternate versions of a query implemented with some of Foxhound's optimizations. In our experiments, these are the P99 approach for (Q1) and the Sketch approach (Q2).

3.8.2 Problem Setup

Next, we describe how we inject problems described in Section 3.1.2, the intuition behind our localization queries, and how the query is implemented by the competing systems.

3.8.2.1 Q1 – Latency Spikes

Random Latency spikes are injected at the x86 servers responsible for processing lookups.

Query Rationale: We instantiate $Q1$ in order to catch the spans with high latencies above the 99th percentile (P99) of the distribution. Entities involved in processing of these spans are generally responsible for the degradation and we want to localize them.

Instrumentation: High latency spans are usually discovered in x86 tracers by post-hoc ordering of all spans by ascending latency and filtering out the ones above the P99 threshold. Foxhound and Jaeger will employ this strategy to profile the fallback lookup span (which is observed at the switch by Foxhound and at the cache server by Jaeger). Optimized Foxhound, however, uses a previous estimation of the P99 (from historic records) to allow the switches to filter out high-latency spans (*i.e.*, at the switch, only export spans for request with latency above a threshold).

For INT, we will use timestamp and egress port to profile per-port traffic patterns. We postprocess INT records to discover a per-port packet-count timeseries. We have found that this instrumentation yields the more consistent and positive results for INT in our evaluations. We repeat the same instrumentation throughout all 3 queries.

3.8.2.2 Q2 – Cache Degradation

NetCache control program failure leads to a stale cache with irrelevant keys.

Query Rationale: Q2 discovers a per-device timeseries of the hit-ratio metric common to caches in general. NetCache normally updates itself by caching “trending” keys to keep the cache relevant and maximize hit-ratio. If the x86 control program of a NetCache switch fails, it will stop updating and the hit-ratio tends to decline as the workload changes.

Instrumentation: The unoptimized Foxhound constructs the hit ratio timeseries by looking at post-hoc span storage and dividing the number of hit spans by the number of miss spans over a sliding window. Jaeger can also discover a hit-ratio timeseries entirely from post-hoc traces by dividing $\frac{\#GetSpans - \#FallBackSpans}{\#GetSpans}$. The optimized version uses Foxhound data structures to maintain a device-local hit-ratio sketch, which is continuously polled to construct the desired timeseries. INT instrumentation is the same as Q1.

3.8.2.3 Q3 – E2E Microservice Failure

We augment DSB's social network (GAN et al., 2019) with in-network compute from NetCache. We then randomly kill components (both x86 servers and switches) from the infrastructure.

Query Rationale: To diagnose a component failure in large service infrastructures, Q3 draws inspiration from Spectroscope (SAMBASIVAN et al., 2011) and profiles the invocation pattern of RPCs inside a large infrastructure (DeathStarBench's Social Network (GAN et al., 2019)). Let Microservices A and B be nodes and calls between them be represented as directed edges. Edge $A \rightarrow B$ has weights proportional to the amount of times A calls B. Q3 searches for changes in any of the edge weights (e.g., a microservice being called more/less than usual). Ideally, after problem injection there will be a noticeable change somewhere in the DAG edges.

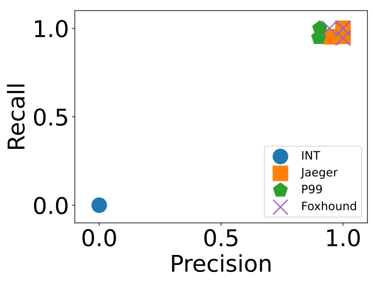
Instrumentation: Because this is a cross-layer query, it is ideal to unite traces from both Jaeger and Foxhound to create the full DAG, this approach is called E2E. For reference purposes, we also run the query with standalone Foxhound and Jaeger. There is no Foxhound-optimized version of this end-to-end query as it is not trivially optimizable through hardware filtering or sketching.

Note on Localization. An important distinction must be made about the localization done in Foxhound. In practice, errors in one service will likely propagate to the calling services. For example, if NetCache fails to answer the microservice that called it, then that microservice will likely also fail and the entire caller-callee chain will recursively fail. This is called an error propagation (EP) chain (GUO et al., 2020b). For the purpose of Foxhound, we consider localization to only be successful when a system localizes the entity in which the initial error (root cause) occurred.

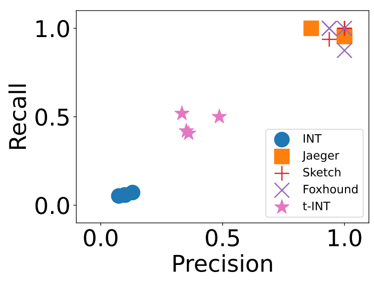
However, while Foxhound focuses on localizing where root causes happened, we do not perform root cause analysis in this evaluation, as the latter generally entails a greater level of detail about why the error happened (usually through manual investigation).

Figure 3.7: Simulated Localization Accuracy

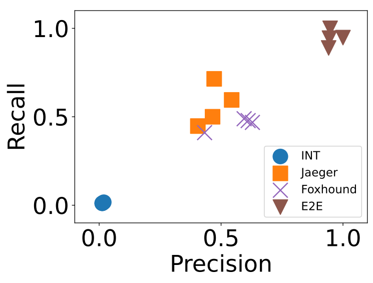
(a) Q1



(b) Q2



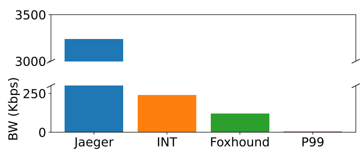
(c) Q3



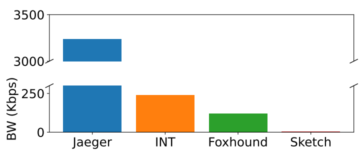
Source: The Authors (2022)

Figure 3.8: BW Sim

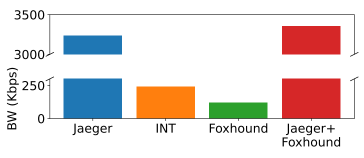
(a) Q1



(b) Q2



(c) Q3



Source: The Authors (2022)

3.8.3 Problem Localization Effectiveness

This section presents case studies to explore the accuracy and overheads of problem localization on a realistic fat tree topology. We ignore hardware limitations and instead simulate an ideal environment in which all the strategies have as much bandwidth as necessary and no packet losses. This allows us to discover the more efficient strategies (*i.e.*, higher in accuracy, lower in bandwidth) and to assess effectiveness of Foxhound optimizations such as hardware filtering and sketching. Due to simulation limitations, our considered rate of requests per second ($O(100)$ RPS) is not representative of high-throughput deployments ($O(100M)$ RPS). Realistic RPS and hardware limitations are discussed in later sections.

Q1 – Localizing Latency Spikes. INT fails completely given that it misattributes all errors to the switches where INT is running. For this query, the latency degradation happens exclusively at x86 servers responsible for servicing fallback lookups. Contrarily, Jaeger and Foxhound were able to localize errors to the x86 server. Jaeger instruments the server process directly and is able to easily discover spans with higher latency at postprocessing.

Foxhound can also diagnose the error. As per Fig. 3.5b, NetCache `READREPLY` spans have latency and source information. The former can be used to identify high latency spans, and the latter will identify the server which processed them.

Finally, our hardware filtering strategy achieved near-perfect accuracy for this scenario (Fig. 3.6a) with lowest overhead (Fig. 3.7a). The slight decrease in accuracy from Foxhound to P99 is due to the fact that P99 does not export all spans, only the ones above a predefined threshold. Specifically, while Foxhound discovers the ground truth post-hoc and is able to accurately pinpoint the high-latency spans, the in-switch filter (P99) is subject to changes during runtime which may cause spans to miss the P99 threshold and be lost. This tradeoff, however, is appealing in terms of bandwidth.

This showcases our strategy of up-fronting postprocessing to trim down the data flow early. By informing our hardware filter with a simple threshold, we can keep tracing targeted to spans of interest, whereas non-filtered strategies will always export spans, dramatically increasing necessary bandwidth.

Q2 – Localizing Cache Degradation

Initially, INT managed to capture some of the anomalous behavior induced by the error. However, the noise from the background traffic ended up hindering the overall accuracy. In general, application-level changes to an INC are not well reflected in traditional network-centric metrics. INT and several other works on network telemetry treat all packets as equal and ignore the semantics of the RPCs.

We created t-INT (targeted-INT) to show that if INT had the ability to discriminate between INC-traffic and non-INC, it could be viable for this specific query as the packet-counts on switch links would be correlated with hit-ratio (the control problem creates lower hit-ratio, which increases packets at the fallback lookup links). Regardless, INT lacks this server-switch coordination and even t-INT is shown to have difficulties when trying to identify INC problem patterns through completely application-agnostic measurements.

Conversely, Jaeger and Foxhound managed to successfully localize the cache degradation. We note that our simulations allow a sampling rate of 100%, which is impractical and sometimes impossible in hardware (as will be shown in later sections). As such, this timeseries-driven query can (and should) be implemented more efficiently by using a Foxhound sketch to perform in-network aggregation. The sketched approach achieved the same accuracy while also being independent of the 100% sampling and utilizing negligible, constant bandwidth (whereas the bandwidth of the other strategies will scale upwards with higher RPS). This result does not argue against the separation of data generation and processing that is customary to distributed tracing, but rather showcases that early processing such as aggregation can be used to dramatically improve the effectiveness of some queries.

Q3 – Localizing Microservice Failures

For Q3, INT suffers from the same traffic isolation problem and the fact that it will always misattribute the errors to switches, even when the error is in x86 services. Additionally, INT was rarely able to localize even the NetCache errors that happened in the switch, since the larger infrastructure of DSB contains a wide range of underlying traffic patterns which generate noise to INT measurements.

As for the compute-centric systems, Jaeger managed to consistently localize x86 errors to their root causes. However, by lack of switch instrumentation, it could not pinpoint in-network problems to the responsible switches. Hence, its accuracy was almost perfect for problems at the x86, and zero otherwise. Foxhound had the exact opposite behavior and managed to localize in-network component failures while completely missing errors from RPCs that did not go through the NetCache INC (e.g., RPCs between two x86 microservices). Lastly, we introduced a new system (E2E) to show that the end-to-end traces which merge x86 spans from Jaeger and INC spans from Foxhound managed to actually localize all types of problems.

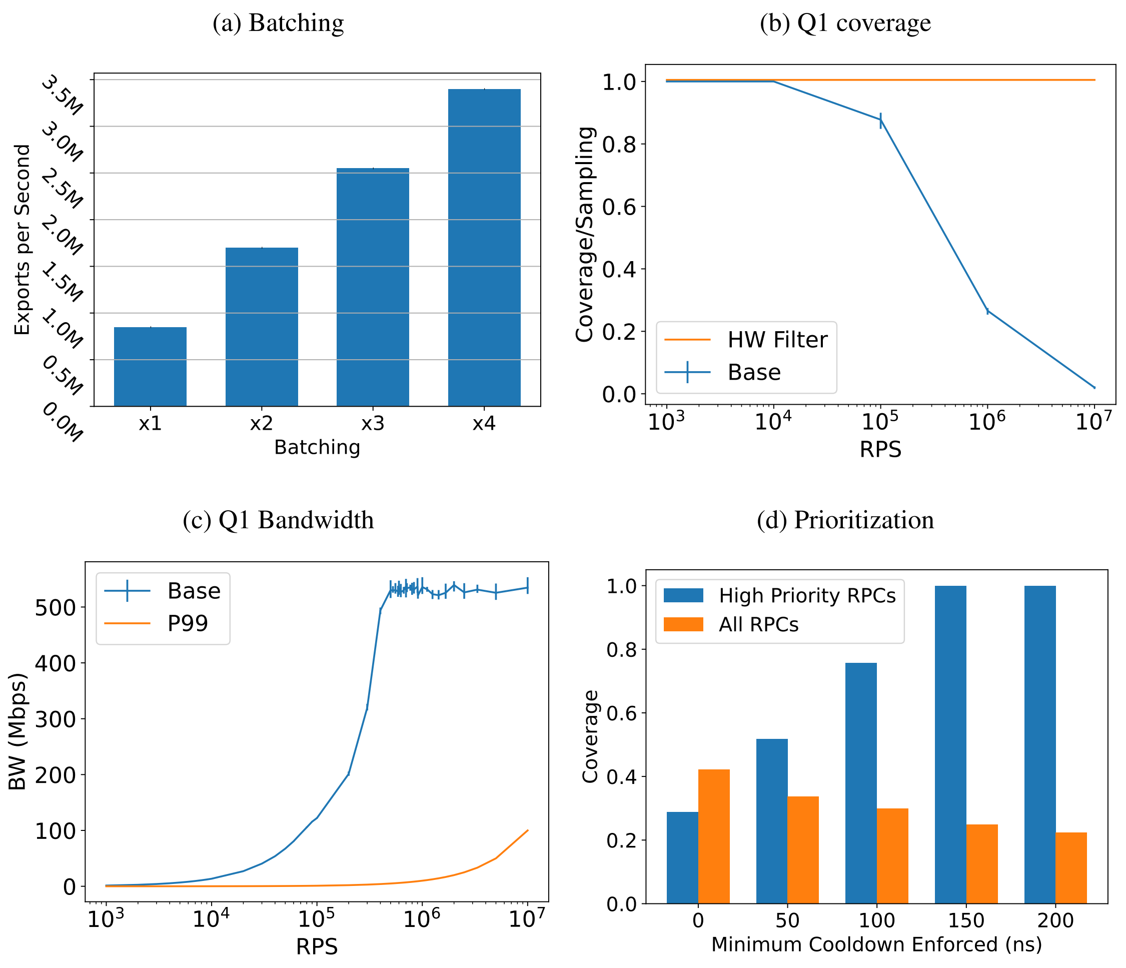
3.8.4 Hardware Micro-Benchmarks

This section presents several microbenchmarks of Foxhound in an Intel Tofino-powered testbed. We showcase the implications of high-RPS tracing and also the benefits of Foxhound optimizations in that context. We also profile hardware-usage of our in-switch modules to understand the practicality of Foxhound in real deployments.

3.8.4.1 Tofino Pipeline Resource Consumption

Table 3.5 shows the resources needed by different configurations of Foxhound and its storage layer. We instrument Foxhound on top of NetCache which serves as the base-

Figure 3.9: Tofino Microbenchmarks



Source: The Authors (2022)

Table 3.5: Tofino Pipeline Resource Occupancy

Version/Resource	**SRAM**	**SALU**	**STAGES**	**VLIW**	**P4 LoC**
NetCache (NC)	25.1%	41.7%	7/12	4.9%	1646
NC + Pipeline Scheduling	25.7%	47.9%	8/12	6.3%	1810
NC + Sched + Batching (x2)	29.1%	54.2%	12/12	7.8%	1940
NC + Sched + Filtering (Q1)	25.9%	50.0%	7/12	5.5%	1849
NC + Sched + Sketching (Q2)	25.9%	50.0%	8/12	6.3%	1840
NC + All of Above	29.3%	56.3%	12/12	8.1%	1977

Source: The Authors (2022)

line. Pipeline stages were the most critical resources[5]. Specifically, we could not compile batching larger than x2 in our P4-14 NetCache codebase due to shortage of pipeline stages. However, this could be due to compiler inefficiencies of the legacy 8.9.1 compiler, as we managed to reach higher batching numbers in a newer compiler, albeit not on NetCache. Lastly, the usage of the other hardware resources both on the table (and others ommited, such as TCAM) was minimal. All circular buffers had 1024 32-bit wide slots and the Q2 sketch had 256 slots.

Batching and PCI limit. To showcase batching larger than x2, we used a P4-16 source code on a newer compiler (SDE 9.4.0). Fig. 3.8a shows that the PCI link bottleneck for Tofino happens at around 800k spans per second and that batching can be used to increase throughput.

3.8.4.2 Hardware Filtering

Hardware filtering is used to optimize Q1. The unoptimized version of Q1 (Fig. 3.8b) illustrates the hardware limitations which were hinted at in previous sections. Specifically, coverage over high latency spans sees a sharp drop after reaching the PCI bottleneck (*i.e.*, max RPS at which PCI can do 100% sampling). The probability of blindly catching interesting spans becomes lower at higher throughputs. Hardware filtering, however, allows for complete coverage over spans with latency above the threshold, keeping bandwidth low even at higher RPS rates (Fig. 3.8c), by only exporting this reduced set of interesting spans.

3.8.4.3 Sketching

Sketching is used to optimize Q2. The unoptimized version of Q2 also suffers from inaccuracies past the PCI bottleneck, as it relies on having a representative set of spans from which it can discover the hit-ratio metric. At higher throughputs, sampling becomes much lower than 100% which hinders the accuracy of Q2. Conversely, the usage of Foxhound's customizable data structures allows the same hit-ratio query to be both perfectly accurate and constant in bandwidth, regardless of the RPS.

[5]In RMT hardware, “usage” of pipeline stages does not stack additively and is dictated by several factors (SIVARAMAN et al., 2016; JOSE et al., 2015).

3.8.4.4 Query Prioritization

For the query prioritization experiment, we ran 2M RPS through the switch, out of which 10k RPS were randomly assigned to a high-priority query. This high priority query was configured to have 10k high priority exports per second. Fig. 3.8d shows that with 150 nanoseconds of minimum PCI cooldown, the coverage over high priority spans was almost total (3x increase against baseline), with a 15% decrease in overall coverage. In raw numbers, the trade-off was that 120k normal spans were lost in order to capture almost all 10k high-priority spans.

3.9 Discussion & Limitations

Integration with Existing Technologies. Despite our proof-of-concept being able to successfully diagnose an end-to-end deployment of an INC-accelerated microservice, we are still subject to limitations of INCs such as incompatibility with TLS encryption[6] or TCP[7]. Those limitations are not specific to INC tracing, but rather to INCs themselves and are being actively discussed and investigated (STEPHENS et al., 2021). We also do not evaluate INCs that rely heavily on packet recirculation. Regardless, we have found Foxhound's instrumentation scheme generalizes to a considerable range of INCs as we have preliminarily instrumented both the P4xos and the CrossPod INCs.

General Applicability of Foxhound Principles. While Foxhound focuses on working around PDP limitation on INC tracing, a few of our insights such as early filtering of spans and data aggregation into sketches could also be used for x86 tracing. Sketch aggregations can be especially useful for providing device-centric runtime distributions, which have been shown to be a useful aggregation mode for tracing data (MACE; ROELKE; FONSECA, 2015). Early filtering of spans also has an x86 counterpart which is tail sampling, where tracers will save a span in case some anomalous condition is met such as high latency. In this regard, tail sampling is currently implemented with fixed policies (GRAFANA, 2020) but we believe our query language could be used to specify and reconfigure these policies.

[6]Programmable switches are not able to decipher the payload of TLS-encrypted packets. This is why INCs tend to use clear, unencrypted formats.

[7]As a general rule, INCs need to parse, for every packet, well-defined application headers in order to work. However, if an application sent those headers using traditional TCP, there is no guarantee that the INC header would not be split between different packets, which would make the header extremely impractical to parse (STEPHENS et al., 2021).

Bypassing the PCI bottleneck. We could export the batched spans directly to the framework through the 100G switch links instead of the switch's PCI-based CPU port. This alternative can achieve higher tracing coverage over the full line rate, however, it is also the most costly in terms of network throughput which does not appeal to production environments. Instead, we argued for and show in our evaluations that the switch's PCI-based CPU interface will offer a quick and low-overhead sink for in-network traces which can be made more targeted and effective through query-specific optimization.

4 SCALABLE IN-NETWORK DISTRIBUTED TRACING — MIMIR

At its core, Foxhound focused on maximizing query expressiveness and data richness. During our evaluation of Foxhound, we noticed promising optimizations, namely (i) keeping statistical distributions as in-network sketches and (ii) using a-priori knowledge of what is relevant to inform early-on hardware filtering, minimizing export overhead. These insights form the base of Mimir, a framework that is focused on maximizing scalability of certain high-impact queries which Foxhound could not scale to more demanding workloads.

We begin this chapter by motivating and recapping the use of aggregation in distributed tracing and INC offload patterns. Finally, we outline domain-specific insights which will guide our design of Mimir.

After our motivation, we sketch our vision and the design choices necessary to accomplish it: offloading large parts of trace processing to the data plane leads to an early and considerable reduction in the data flow. Mimir essentially wants to limit data flow by only capturing data summaries. While this limits flexibility (e.g., not all Foxhound queries can be implemented in Mimir), it will also allow PDP tracing to scale to more demanding workloads. Lastly, we decompose the architecture and prototype of Mimir and finalize with its evaluation and discussion of central topics.

4.1 Motivation

In this section, we discuss crucial attributes of distributed tracing (Section 4.1.1), describe recent efforts to offload distributed applications and the limitations of extending distributed tracing to them (Section 4.1.2).

4.1.1 Distributed Tracing and Aggregation

Distributed tracing is currently the dominant method for localizing performance problems in distributed systems. Many hyperscalers (GOOGLE, 2021; ALIBABA, 2021; GUO et al., 2020a; KALDOR et al., 2017) and startups (PINTEREST, 2022; NETFLIX, 2022; KAMON, 2022) use it to capture structural and temporal properties, which engender a deeper understanding of their distributed systems. This deeper understanding

Table 4.1: Tracing Adopters

System	**Company**	**Outlier Detection**	**Structural Aggregation**	**Temporal Aggregation**
Cloud Trace (GOOGLE, 2021)	Google	Req./Aggr.	–	✓
Tracing-Analysis (ALIBABA, 2021)	Alibaba	Aggregates	–	✓
PinTrace (PINTEREST, 2022)	Pinterest	–	–	✓
JCallGraph (LIU et al., 2019)	JD.com	Requests	–	✓
Jaeger (JAEGER, 2020)	Uber	–	✓	✓
GMTA (GUO et al., 2020a)	Ebay	Aggregates	✓	✓
Canopy (KALDOR et al., 2017)	Facebook	–	✓	✓
Inca (NETFLIX, 2022)	Netflix	Requests	✓	✓
Kamon Telemetry (KAMON, 2022)	Kamon	–	✓	✓
Lightstep (LIGHTSTEP, 2022)	Lightstep	Req./Aggr.	✓	✓

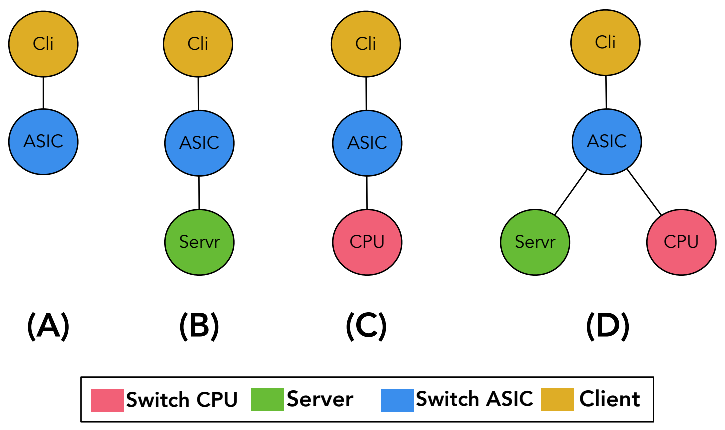
Source: The Authors (2022)

allows for localizing and pinpointing performance problems to specific processes or services within their infrastructure. At its core, distributed tracing captures the flow of execution for a request across the processes that handle the request. Each process generates data, called *spans*, when it handles a request: a span captures start and end processing times. The collection of spans generated by all processes which handle a specific request is called a trace. Recall from Chapter 2 that a trace is essentially a DAG of spans which happened during the processing of a request.

To do perform diagnostic activities, *traces* are often aggregated into "groups" based on clustering techniques or request type. This aggregation enables operators to understand how their infrastructure has evolved by analyzing changes in clusters over time – we call this *system-level performance analysis*. Alternatively, they use the aggregated information to detect real time performance anomalies by finding individual traces in a cluster that differ from others (i.e., outliers performance) – we call this *request-level performance analysis*. Table 4.1 demonstrates the types of aggregation-based diagnosis supported by current infrastructures. The key motivating observation of our work is that aggregation is a dominant method for understanding and diagnosing performance problems both at the request level and more globally at the system level. Below we elaborate on two dominant problem diagnosis techniques and discuss how various aggregation approaches are used.

- **Temporal Aggregation-based Diagnosis.** allows operators to detect problems due to resource sharing or interference between different functions running on a node. These methods focus on inferring the expected number of concurrent requests or expected resource usage. Anomalies, either too many or too few requests, are considered indications of problems.

Figure 4.1: NetCache Operation Diagrams



Source: The Authors (2022)

- **Structural Aggregation-based Diagnosis.** detects problems due to application processing logic (e.g., long processing latencies or incorrect logic). These methods focus on inferring the expected latencies or expected application behavior (e.g. call structure). Here also, anomalies, unusually high latency or unexpected behavior are considered indicative of problems.

While both approaches can be used to diagnose structural or request-level problemswith unexpected behavior, existing networking techniques (WANG et al., 2020b; BASAT et al., 2020b; KIM et al., 2015) which excel at only diagnosis queuing problems or network latency problems fall into the former class (Temporal). Interestingly, for the latter class which requires introspection into program code and a broader understanding of interactions between different components (*e.g.* network switches and x86 servers), there is a need for new techniques.

4.1.2 Common Design Patterns of In-network Compute (INC)

With the advent of programmable data planes, a vast range of distributed applications have been offloaded into the network. These offloads generally follow one of a few predefined patterns. Next we illustrate these patterns with the help of the previously discussed NetCache INC (recall INC Case Study – NetCache in Section 3.1.1).

NetCache accelerates traditional caches (e.g., MemCached or Redis) by storing a subset of key-value pairs at ToR switches, thus the ToR switch can respond to “GET” requests for the key-value pairs that it currently stores. To realize this vision, NetCache must identify the most frequent keys being queried and cache them. To do this, NetCache maintains a data structure in the ASIC to summarize the current lookup patterns and NetCache runs a process on the switch CPU to periodically analyze this data structure and appropriately update the ASIC cache.

Design Pattern #1: As with NetCache, many other INCs run in a *hybrid mode* with aspects compiled to the ASIC and an accompanying process running on the switch CPU (Table 4.2 (Col 1)).

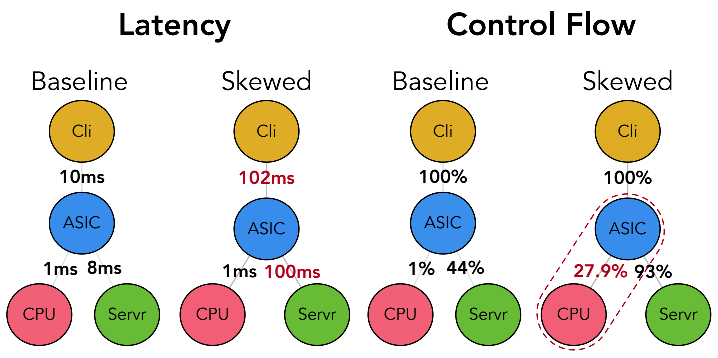
Design Pattern #2: As with NetCache, many other INCs typically rely on the application running on the x86 as the last resort when the INC cannot process an RPC request either due to lack of data or because the functionality is not offloaded – we call this a *Fallback pattern* (Table 4.2 (Col 2)).

Design Pattern #3: In the case of NetCache, the functionality being offloaded is client facing and thus clients directly interact with the INC. In other scenarios, e.g., ML (LAO et al., 2021), Coordination (YU et al., 2020), the functionality being offloaded is an application internal functionality then only servers of that service interact with the INC. Thus, while the servers in the Web-Tier will interact with the Cache Tier's INC (i.e., NetCache), only members of the DB Tier will interact with the DB Tier's INC (i.e., NetLock). The key observation here is that the location for inserting an INC's span data for reconstructing a *trace* is highly dependent on the nature of the functionality being offloaded.

Design Pattern #4: Finally, the purely event-driven nature of programmable data planes disaggregates this workflow into several distinct control flows, e.g., while the cache

Figure 4.2: Span DAGs used to localize problems.

(a) **Tail Latencies** (b) **Control Flow Distributions**



Source: The Authors (2022)

Figure 4.3: Annotated NetCache Code

```
 1  #DevOp Annotated NetCache.p4
 2  if(pkt.op==WRITE)
 3      CacheWrite(pkt.key,pkt.val);
 4      @End("HotOut")
 5
 6  if(pkt.op==READ)
 7      if(isCached(pkt.key)) #Hit
 8          @Begin("Hit")
 9          pkt.val=CacheRead(pkt.key);
10          ipv4.dst = Sender;
11          @End("Hit")
12      else #Miss
13          @Begin("Fallback Lookup")
14          ipv4.dst = Server;
15          if(HotKey(pkt.key))
16              @Begin("HotOut")
17              CloneToCPU=True;
18
19  if(pkt.op==READREPLY)
20      @End("Fallback Lookup")
```

Source: The Authors (2022)

Table 4.2: Offload Patterns for Different INCs

INCs	Hybrid	Fallback	Functionality	Event-driven
NetCache	Yes	Yes	External	Yes
P4xos	Yes	No	External	Yes
CrossPod	No	Yes	Both	Yes
DAIET	No	No	Internal	Yes
ATP	Yes	No	Internal	Yes

Source: The Authors (2022)

hit (a) is captured as a continuous workflow (Figure 4.3[1] lines 8-11) the HotOut is two distinct workflows (Figure 4.3 lines 2-4 and lines 16-17). This is distinct from traditional x86 servers where request processing follows a run to completion model and context is maintained between calls.

Takeaways: The NetCache use case illustrates several dimensions along which INCs can be characterized: These dimensions also incidentally highlight complicated interactions that are themselves potential avenues for performance problems. In Table 4.2, we present a list of INCs and characterize them along these dimensions. Effectively in-

[1] In this figure, we showcase our disaggregated `@BEGIN` and `@END` annotations. We found it necessary to have this disaggregation in order to be able to better model application semantics in the context of these event-driven programmable devices (e.g., it allows us to match a request to its reply despite the switch not keeping any context over the packets).

tegrating traces from programmable data planes with traditional distributed tracing needs to capture and address the relationships between the INC and the server applications.

- INCs inherent event driven nature requires maintaining and tracking state between disjoint events to recreate spans. Yet, maintaining state locally on a switch introduces scalability challenges (Patterns #4).
- INCs encapsulate a range of application-specific functionality. Yet, their inclusion into distributed traces must introduce edges and nodes that accurately reflect the flow of events between the accelerator and the applications (Patterns #1 and #3).
- INCs are often split across switch ASIC and switch local CPU. Yet, tracing must accurately capture this Hybrid pattern by tracing both ASIC executions and also spin-off x86 executions which may happen in the switch CPU (Pattern #2).

4.2 Design

We envision that effectively disaggregating distributed tracing and localizing problems to programmable data plane requires solving three challenges: (1) overcoming hardware memory constraints to *scalably maintain sufficient execution context to create spans*, (2) overcoming hardware processing constraints to *support general aggregate-based diagnosis of trace data*, and (3) overcoming language restrictions to *provide a customizable set of outlier and anomaly detection algorithms*.

To address these challenges, Mimir builds on insights about the common case diagnostic workflow and application design patterns. Specifically, Mimir explores a novel partitioning and delegation of diagnostic functionality to the switch data plane, which explores a judicious decoupling of storage and processing across different components of the data plane. Next, we discuss the design principles that enable Mimir to solve these challenges and highlight the key observations that underpin our principles.

First, while observability frameworks support an arbitrary set of analyses, in reality (MACE, 2017; CHOW et al., 2014; KALDOR et al., 2017), DevOps diagnosis efforts are mostly centered around limited set operations (e.g., latency, control-flow). Second, today, developers annotate code (CHOW et al., 2014; KALDOR et al., 2017) to provide existing telemetry systems with visibility into their programs. Mimir capitalizes on these observations and proposes the following distributed in-network design approach. [2]

[2]Note that Mimir creates an RPC abstraction for DevOps to express the complex interactions of their

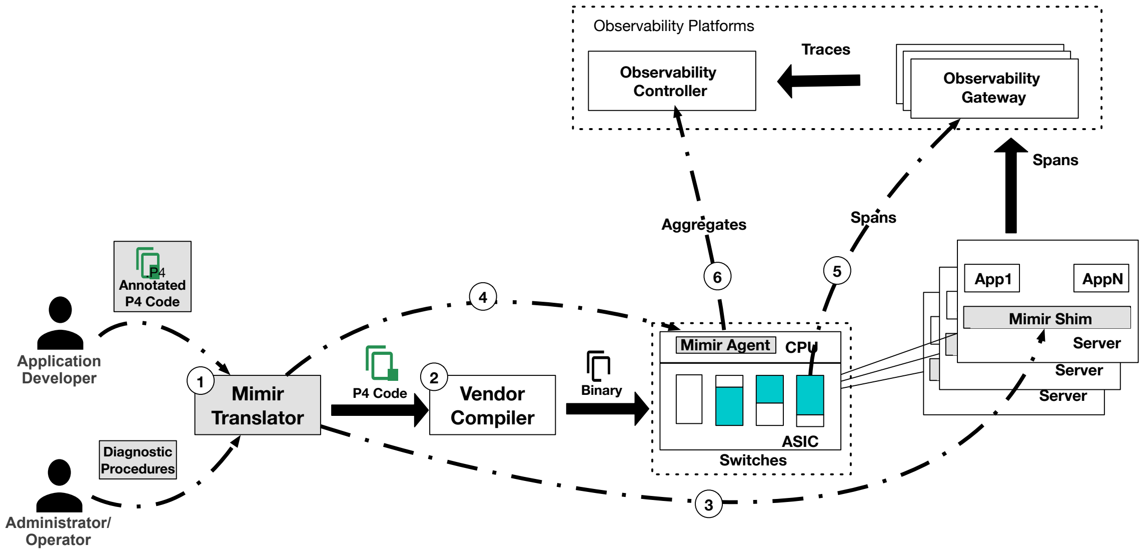
To scalably maintain sufficient execution context to create Mimir **Decouples Transient from Persistent storage.** Typically, trace creation requires temporary storage of per-RPC data, whereas trace analysis requires the storage of aggregated data. While storing per-RPC data on a switch is not scalable, we observe that span creation requires the switch to process both requests and responses. Developer annotations explicitly identify relevant content on the request and response processing code paths. Mimir builds on these observations to design a storage-layer which scales by decoupling storage of transient per-RPC data from persistent aggregate data by storing per-RPC data in packets and aggregate data in customized in-switch data structures (e.g., histograms). The use of packet headers to store per-RPC data allows our storage layer to scale to large workloads.

To support general aggregation-based diagnosis, Mimir **Delegates performance diagnosis.** Existing trace-based diagnostic efforts compare individual traces (actual behavior) against aggregated ones (expected behavior). Unfortunately, programmable data planes cannot export sufficient data to a centralized location for such analysis. Mimir builds on the observation that the comparisons are often made pairwise by comparing equivalent nodes and edge distributions in the expected behavior (e.g., aggregated behavior) against the nodes in the actual trace. Instead, Mimir delegates these pairwise comparisons to the switch. In particular, instruments programs to store the aggregate data corresponding to the delegated portion of the pairwise analysis and also instruments them to perform the predetermined analysis.

To provide a customizable set of outlier and anomaly detection algorithms, Mimir **Tiers Performance Diagnosis** Empirical evidence shows that operators are interested in two broad classes of trace-based analysis: first, identifying when a request's behavior differs from the system's expected behavior, and second, detecting when a system's general behavior differs from its expected behavior. However, the comparison of aggregate system behavior requires complex algorithms (e.g., Earth Mover's (VASERSTEIN, 1969; PANARETOS; ZEMEL, 2019) or distribution comparions (MASSEY, 1951; SMIRNOV, 1944)), the request-based analysis are performed using simple thresholds (e.g., latency thresholds, number of requests), which can be easily implemented in line rate in hardware. Mimir leverages the differences in granularity of the two diagnosis efforts to explore a tier-ed design system wherein the fine-grained per-RPC analysis, which can be implemented using lightweight threshold, are implemented in switch's ASIC while the coarse-grained system-level analysis, which requires complex analysis, is implemented

offloaded applications and coordinate this abstraction using minimal state on packet headers.

Figure 4.4: Mimir Workflow



Source: The Authors (2022)

on the switch's local CPU.

4.2.1 Workflow

Mimir requires developers to introduce annotations and operators or SREs to specify the list of performance problem of interest (e.g., the latency percentile of interest or the nature of aggregate system-level behaviors changes of interest). Naturally, manually annotating INCs introduces some overheads we note that today programs are either manually (CHOW et al., 2014; KALDOR et al., 2017) annotated to enable distributed tracing. We support a subset of the generaly annotations (In Table 4.3) used for creating basic Spans and traces (a Mimir-annotated example of NetCache is found on Fig. 4.3).

In Table 4.4, we provide a brief description of the set of performance procedures supported for defining the performance problems of interest. Our system-level and request-level procedures are a union of analysis used in academic (SAMBASIVAN et al., 2011) and industrial systems (OPENTELEMETRY, 2021) for trace-based diagnosis.

Mimir takes the annotated source code, and a set of performance problem definitions, and uses these as inputs into **Mimir's Translator** which automatically instruments the P4-code with sufficient primitives to capture spans, store aggregated data, and perform request-level performance diagnosis (1). Subsequently, Mimir uses a traditional

Table 4.3: Annotation primitives

Annotation	Description
`@Begin(Span,<Parent>)`	Starts an in-network span
`@End(Span)`	Ends an in-network span

compiler to compile the instrumented P4 code and uses a traditional toolchain to deploy the compiled code to the switch ②. Mimir configures the **Mimir-Shim** to capture RPC information from packets exchanged from clients ③. Finally, Mimir configures and parameterizes the deployment, e.g., configuring the **Mimir-Agent** running on the switch CPU to perform outlier/anomaly detector functions on the appropriate aggregate data and installing rules to setup thresholds ④.

Table 4.4: Anomaly Detection Procedures.

API Call	Diagnostic Level	Description
`Latency(Percentile)`	Span	Capture spans where latency is greater than a %.
`ControlFlow(Thres)`	Span	Capture spans where control flow is outside a threshold.
`Distribution(Algo, Thresh)`	Aggregate	Create an alarm when a distr. is skewed from a baseline. Op. specifies comparison algo for detecting skew

Source: The Authors (2022)

4.2.2 End-to-End Diagnosis

To achieve a seamless end-to-end diagnosis of distributed systems, Mimir integrates into existing distributed tracing frameworks, e.g., Jaeger (JAEGER, 2020). Recall, in Jaeger (with tail-sampling enabled), individual servers export spans to a Gateway, which recreates traces, filters significant traces, and exports important traces to a centralized data store (the controller). Mimir generates information which it sends to the Gateway or to the centralized data store. Specifically, when individual spans violate operators-specified thresholds (i.e., request-level performance problems), then Mimir generates and sends data to the Gateway, which triggers Jaeger to capture the entire trace for detailed analysis and export them to the controller ⑤. On the other hand, when aggregate trace behavior violates operator specified characteristics (i.e., system-level performance analysis), then Mimir exports summarized behaviors to the controller and merges them into the aggregate traces used for posthoc analysis of system behaviors $\textcircled{6}$.

4.3 Mimir Architecture

To understand how the different components within Mimir operate, we begin by illustrating, in Figure 4.5, Mimir's high-level functionality and interaction of its core components. When distributed applications generate RPC requests (*i.e.*, RPC caller), Mimir's shim layer intercepts and introduces tags to the RPCs for identifying and correlating packets for tracing at switches $\textcircled{1}$. These requests are forwarded into the network $\textcircled{2}$ and when an INC processes an RPC, Mimir adds a context to the request $\textcircled{3}$. At the receiving server (*i.e.*, RPC responder), the associated shim layer intercepts and extracts the context of the receiving RPC for further processing $\textcircled{4}$. When the response is generated, the shim layer adds the context again[3] before forwarding the response into the network $\textcircled{4b}$. When the INC gets the RPC response, it removes the context and uses it to create a span, stores the latency profile and execution control flow into appropriate aggregate structures $\textcircled{5a}$, performs request-level analysis by comparing the span data against pre-defined thresholds $\textcircled{5b}$, and in case an anomaly is detected the offending span is exported to the observability gateway $\textcircled{5c}$. Finally, the RPC response is forwarded to the RPC calling server. In the background, Mimir periodically exports the aggregates to the local switch CPU $\textcircled{6a}$, where pre-specified outlier and anomaly detection procedures run. This aggregated information is further exported to the tracing collector $\textcircled{6b}$. Next, we discuss the detailed functionality of the above system components.

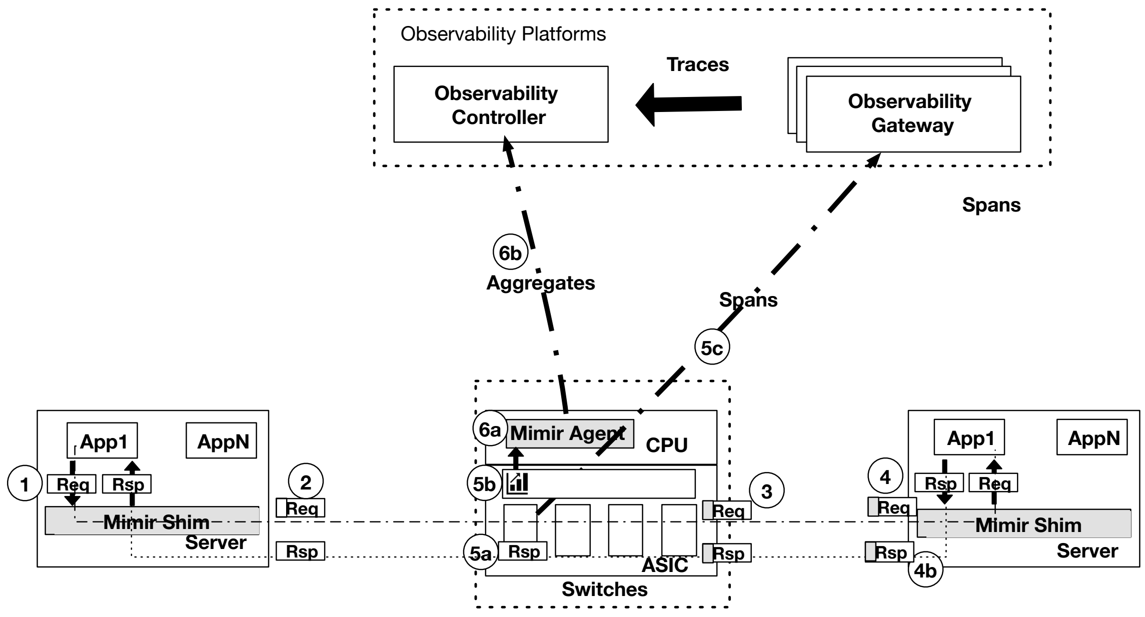
4.3.1 Decoupling Storage with *Translator*

The *Translator* takes as input an annotated P4 program and operator specified queries (Table 4.4) and automatically instruments the P4 program to include: (i) our tailored data structures for capturing and storing the aggregated observability data, (ii) processing logic to export temporary span data into request packet headers and to subsequently retrieve this data from response packets, and (iii) processing logic to perform request-level analysis and generate span alerts to the gateway.

Aggregate Storage. Mimir's *Translator* must instrument the code to use simple

[3]Request and responses are matched by RPCID.

Figure 4.5: Mimir Architecture



Source: The Authors (2022)

data structures to support aggregate-style analysis of the latency or the control flow. In particular, these data structures must be small and linear due to the hardware constraints, i.e., limited registers arrays. Also, they must be easy to update to address language/processing constraints, i.e., limited processing cycles and processing logic. We opt for a design that explicitly trades off simplistic update semantics with a complex query mechanism. This trade off aligns with the distinction between switch ASIC, where updates are performed, and switch local CPU (or centralized controller), where complex anomaly detection (or central analysis) is performed.

Our *Translator* introduces array-based histograms (CORMODE; HADJIELEFTHERIOU, 2010) and adapts prior work in the context of Mimir. The main reason why we opted to use array-based histograms over probabilistic counting sketches is because while sketches allow us to further increase the number histograms we can support, we found that these probabilistic structures require added control logic, reduced accuracy, both of which we avoid by using pure exponential bins. More importantly, because the cardinality of our data (e.g., number of exponential bins for latency and distinct control flows) is small, the benefits of probabilisitics structures are significantly smaller than in traditional networking scenarios with high cardinality (e.g., number of distinct flows).

Extracting Histograms From Annotations: Our *Translator* walks through a pro-

Table 4.5: Mimir P4 Templates

Template	P4 LoC	Parameterization
Latency Set	7	Annotations
Latency Calc	16	Annotations
Latency Histogram	8	–
Control-Flow Set	7	Annotations
Control-Flow Hist.	16	–
Span Trigger	34	Procedures (Table 4.4)

Source: The Authors (2022)

gram replacing annotation that indicate the start of a Span (i.e., `@Begin`) with templates ("set" in Table 4.5) that add context into meta-header fields and annotations that indicate the end of a Span (i.e., `@End`) with templates that calculate and increment the appropriate bin in the histogram.

Domain-Specific Histogram Optimizations: Our work builds on two key insights about the nature of our histograms to further optimize our data structures.

First, the latency histograms are sparse, in-nature. Rather, than exploring equal sized bins, we use exponential sized bins which simultaneously allow high accuracy and low overheads (TRAEGER; DERAS; ZADOK, 2008; JOUKOV et al., 2006). Unfortunately, existing hardware does not support appropriate operation to support exponential or logarithmic operations. Instead, we allocate and use a TCAM table to mimic the logarithmic operations. To approximate the binary logarithmic operation, we perform a TCAM lookup to determine the most significant bit of each timestamp (which is guaranteed to yield the binary logarithm of any timestamp t, truncated down to the nearest integer). This TCAM table is prepopulated to ensure that this lookup returns the index for approprimate bin. We note that this logarithm lookup table can be shared by all latency histograms. Finally, the granularity of the latency histogram can be manipulated through a DevOp-specified configuration value (e.g., shifting from log base 2 to a different base).

Second, for control-flow histogram, we observed that while there are an exponential number of potential control paths, the number of valid combination of control paths is small. We can identify them by using Ball-Larus algorithm in PDPs (KODESWARAN et al., 2020) and this will restrict the number of bins. Together these allow us to restrict the number of bins in a control-flow histogram and to reduce the size of our data structures.

Third, and more generally, we note that given the fact that the size of these histograms are generally smaller than the size of a register array, we have been able to multiplex multiple of these histograms into arrays.

Temporary Storage. To reconstruct a *span*, distributed applications usually maintain context to track the flow of execution across function calls. However, programs maintain little contextual state in programmable data planes due to memory constraints. In Mimir, guided by developer annotations, the *Translator* instruments the program to generate appropriate context information and store them in packet headers. Specifically, we focus on the context required to determine span latency and identify span control paths: start-time and SpanID. We plan to explore a richer context that captures event information as part of future work.

4.3.2 Tier-ed Analysis with *Mimir-Agent*

Mimir supports two diagnostic levels: request-level (on switch ASIC) and system-level (on switch CPU).

System-Level. System-level procedures compare the current distribution with prior ones to detect changes in system behavior. Examples are comparing the latency distribution to see if it has increased (e.g., more lengthy RPCs) and comparing the control-flows to discover changes to the request pattern (e.g., a relative increase in misses). Our goal is to support a broad range of procedures used in prior-works (TRAEGER, 2008; TRAEGER; DERAS; ZADOK, 2008; JOUKOV et al., 2006; SAMBASIVAN et al., 2011). We currently support Earth Mover's Distance (EMD) algorithm, a well understood and wildly used diagnostic algorithm for understanding changes in the distributions. We select EMD because unlike other algorithms it does not suffer from overestimating the importance of comparing one bin to its neighboring bins (cross-bin algorithms which compare bins all-to-all) or underestimating it (bin-by-bin algorithms which only compare two by two) (TRAEGER, 2008; TRAEGER; DERAS; ZADOK, 2008; JOUKOV et al., 2006). Given that these procedures run on switch-CPU in the *Mimir-Agent*, we can easily extend the *Mimir-Agent* to include other relevant procedures.

Request-Level. Conversely, Request-level diagnosis is used for finding anomalous spans. Request-level procedures include finding spans with latency at the tail-edge of the distribution (e.g., P99) and unlikely structure, e.g., less than 1% observed occurrence. These values, such as P99, can be discovered separately at each switch by using information from Mimir distributions. We support these diagnostic procedures in the switch ASIC. For this subset, our *Translator* instruments the code with one table for holding threshold values for each of the appropriate checks (latency greater than a threshold or bitmap matches specific bitmaps) and appropriate code for generating a clone packet which sends the span information to the gateway.

4.3.3 Mimir-Shim

Our shim layer is responsible for inserting and deinserting the Mimir header used to capture RPC information – this allows for Mimir to interoperate transparently with existing distributed systems. Essentially, before packets are sent, a shim-layer at the caller encapsulates packets by inserting a blank Mimir header with the fields necessary for the switches to record compute data (e.g., timestamps for each RPC). When packets are received, the shim-layer at the receiver removes Mimir headers, in case they are present, to allow transparent sending and receiving by applications.

4.4 Prototype

***Translator*:** The Mimir P4-to-P4 translator is written in 1100 lines of Python Code. We fill the templates by using Python F-strings, which allows our program to run compiler logic inline with P4 code (similarly to templates of BeauCoup (CHEN et al., 2020)). All P4 templates which are used to instrument the storage layer for annotations are written in a combined 88 lines of P4-14 and 198 lines of Python. All of our exponential bins histograms (for latency) use 32-bit timestamps and index the log_2 of that timestamp (through our approximation algorithm, which uses corresponding 32 TCAM LPM rules) within a 32-slot wide register. All of our P4-Tofino compilations used Barefoot SDEs 8.9.1 (legacy) and Intel P4 SDE 9.4.0. We discover Tofino pipeline resource usage by using the P4insights tool from SDE 9.4.0.

Mimir-Shim* and *Mimir-Agent: The shim layer and switch-CPU agents are written in a total of 2422 lines of Python Code. To read/write to stateful switch elements, our switch-CPU agent uses either the PCI-based API of the Tofino ASIC (for a hardware deployment) or the Thrift API of BMv2 (P4LANG, 2020) (for emulated topologies). For deploying our emulated testbed, we extended mininet (LANTZ; HELLER; MCKEOWN, 2010). For our hardware testbed we used tc. For our E2E deployment, we instantiate the social network microservice of DSB (GAN et al., 2019) using Docker (MERKEL, 2014).

In all of our deployments we use a 1 second interval between polling the switch distributions (1 poll per second), but early experiments suggest that much higher frequencies are attainable using the Tofino API (1M polls per second).

Collectors: For end-to-end tests, we use the Jaeger collector (JAEGER, 2020) and leverage Jaeger's JSON endpoint for integration. For standalone Mimir experiments, we use an unoptimized Python collector (710 LoC).

INC-related Setup: We extended the same client and server applications from both open and closed-source versions of the three evaluated INCs. All of them were written in Python and were easy to adapt with our Python shim-layer.

4.5 Evaluation

We evaluate Mimir's performance characteristics and general feasibility with a Tofino-based hardware testbed and Mimir's scaling properties with a large-scale emulated testbed.

4.5.1 Experiment Setup

We begin by describing our setup.

Testbed Setup: Our hardware testbed comprises 2 servers with 8-core Intel Core i7-9700 CPU @ 3.00GHz, 16GB RAM, running Ubuntu 18.04 with kernel version 4.15. The servers are connected through 40Gbps Agilio LX SmartNICs to a Wedge 100BF-32X 32-port programmable switch powered by a 3.2Tbps Tofino ASIC and a quad-core Intel D1517 x86 CPU @ 1.6GHz. For NetCache experiments, we use 1 server as the client and the other as the server. Our workload uses the same distribution as the NetCache paper (JIN et al., 2017).

Applications (INCs): We evaluated Mimir using three INCs which showcase a diverse set of designs for in-network computing. **NetCache** described earlier (Section 4.1) illustrates a design used in several subsequent works (JIN et al., 2018; YU et al., 2020; LI et al., 2020). **P4xos** offloads the Paxos consensus protocol (LAMPORT, 1998) into network switches, avoiding the usual bottlenecks of the software implementation and drastically increases the maximum throughput. Finally, **CrossPod** encompasses different compute blocks such as in-network MemCached acceleration and compression, each with its own offload patterns.

Emulator Setup: We use the BMv2 software switch (P4LANG, 2020) to emulate a $k = 4$ fat-tree topology in all our experiments. The experiments were executed on an Ubuntu 16.04 virtual machine running with 4 cores @ 2.9GHz and 16GB of memory. For **NetCache** (JIN et al., 2017), we divide the hosts into two groups: hosts in two of the pods act as clients, and send requests to servers in the other pods. We deploy a NetCache instance on the ToR switches of the server pods and attach an additional host to act as the switch CPU. For **P4xos** (DANG et al., 2020), in each pod we assign a host to each role: client, proposer server, and two learner replicas. We also assign hosts as an in-network coordinator (aggregator layer) and as an in-network acceptor (edge layer). For **CrossPod** (SULTANA et al., 2021), hosts in one pod transmit data to a server in another pod. In between them is CrossPod's MemCached booster (in-network cache). Unlike NetCache, this time, the cache is deployed on the client pod's ToR.

Metrics: For *accuracy*, we focus on singling out the offending components (i.e., *localization*). We run several 10-second simulations, and at $t = 5$ seconds, we introduce specific failures (described next). Given the ground truth about the type of problem(s) injected and the location of injection, we can easily determine if detection and localization are correct or incorrect. For *overheads*, we focus on Mimir's switch resource and network bandwith footprint.

Workload: For NetCache and CrossPod, we reuse the same workloads and or flow characteristics as the original papers (JIN et al., 2017; SULTANA et al., 2021). For P4xos, we randomize the keys and values that each client sends to proposers. [4]

INC Failure Injection. Bugs in programmable data planes are increasingly diverse and prominent (DUMITRESCU et al., 2020; TIAN et al., 2021; STOENESCU et al., 2018; FREIRE et al., 2018; LIU et al., 2018; KANNAN et al., 2021). We take inspiration from previous works to extract representative bug categories that demonstrate Mimir's ability to localize problems, namely:

Latency Degradation (`LAT`*) (KANNAN et al., 2021; WANG et al., 2020b):* We artificially increase the latency of RPCs by having the server randomly pause for a predetermined period when responding to RPCs requests.

Misconfiguration (`CFG`*) (DUMITRESCU et al., 2020):* We alter the INC's source code to misconfigure their data structures, by specifying data structure sizes that are too small for their workloads. For example, with *NetCache*, we misconfigure the sketch size

[4]For CrossPod, Because of emulation limitations (BMV2. . . ,), all aggregate flow rates are set at approximately 100 RPS.

which leads to constant frequency overestimations and ultimately false-positive hot-outs.

Control Failure (`CTRL`*) (TIAN et al., 2021):* We terminate the INC's control program which is running on the switch's local CPU.[5]

Versioning (`VER`*) (TIAN et al., 2021):* We randomly select a single switch to run an outdated INC which uses headers with different fields (*P4xos*) or header values (*Cross-Pod*). Due to these header changes, the INC may ignore packets (no optimizations performed) or process them incorrectly (matching to a garbled header value).

4.5.2 Competing Systems & Instrumentation

We compare and evaluate Mimir against the following three classes of observability techniques:

AppTracing. (JAEGER, 2020; ZIPKIN, 2021; FONSECA et al., 2007), *e.g.*, Jaeger, collect traces from x86-applications and stores them at a central collector. We instrument the applications with annotations, *e.g.*, for NetCache we annotate the client and cache server. We reuse Spectroscope (SAMBASIVAN et al., 2011) localization techniques to detect problems.

Strawman. is a barebone, unoptimized version of Foxhound which collects spans from the entire in-network execution of a request. This approach generates a packet for each INC invocation and exports whatever information is available at that time to a central collector. For localization, *Strawman* reuses the same procedures as Mimir (Table 4.4).

4.5.3 Problem Diagnosis (Localization)

We begin by analyzing the accuracy of Mimir against competing techniques in three scenarios:

Emulator Deployment. First, we consider an emulated deployment in which INCs accelerate a two-tier services. In Fig. 4.6, we observe a bimodal behavior with Mimir and *Strawman* correctly localizing all of the in-network problems expect the latency problem (i.e., LAT Fig. 4.5a) which was a problem at the server. Moreover, we

[5]For NetCache, this prevents the ASIC cache from being updated, leading to a stale cache. For Cross-Pod, this causes compute blocks to become unreachable. For P4xos, this prevents the configuration of multicasting, resulting in traffic circumventing the acceptor switch and in essence preventing traffic from using P4xos.

observe that *AppTracing* is able to correctly localize only the latency problem (i.e., LAT Fig. 4.5a). We note that this behavior is consistent because both Mimir and *Strawman* are only using in-network information whereas *AppTracing* is only using server-side information. Later, when we integrate Mimir with *AppTracing* for an End-to-End system on our Testbed (Fig. 4.6a), the combined technique is able to localize these latency problems.

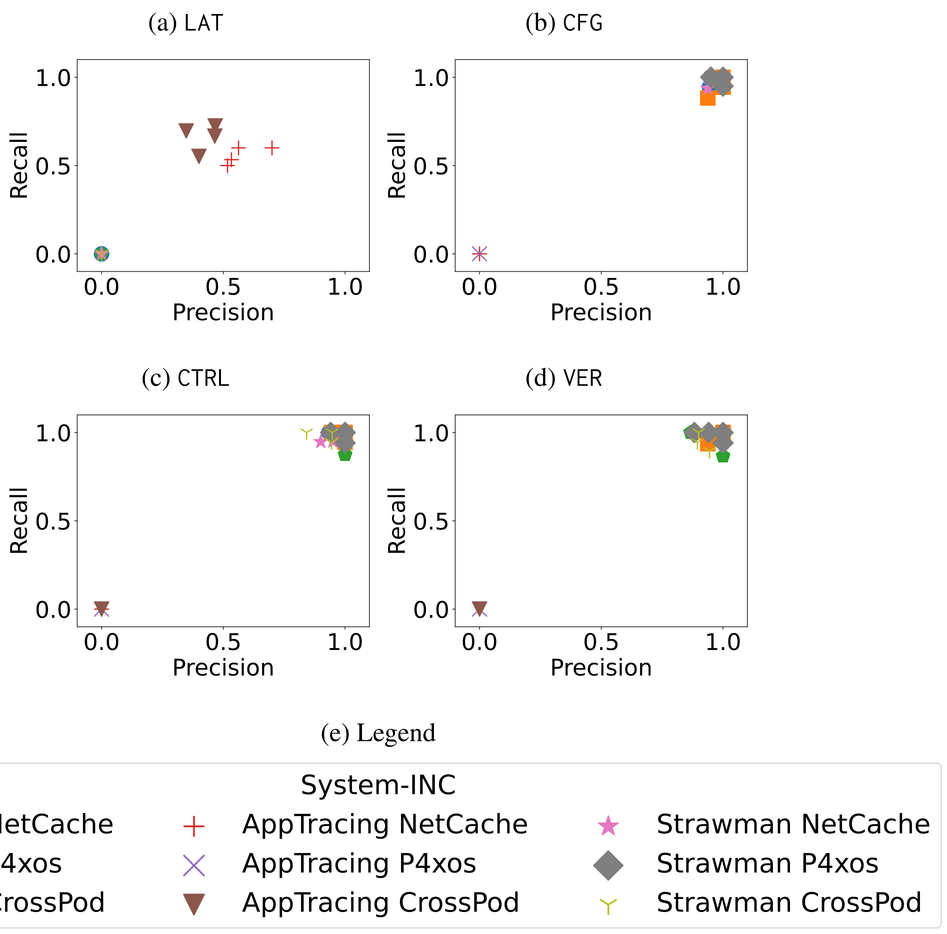
Additionally, we note that in these experiments throughput is sufficiently low for *Strawman* to capture sufficient information however we show how *Strawman*'s accuracy degrades at scale in (Section 4.5.3).

Physical Switch. In Fig. 4.6a, we demonstrate with our hardware testbed that the accuracy observed in the emulator for Mimir is representative of that from an actual physical switch. Moreover, while Mimir retains a high accuracy, we observe that, at these higher rates, *Strawman*'s accuracy degrades. In analyzing *Strawman* (Section 4.5.4), we observed that at high data rates *Strawman* is unable to export spans, which leads to low coverage and ultimately nullifies accuracy. Finally, we note that the latency misattribution is still present for Mimir.

End-to-End Deployment. Finally, we focus our analysis on the `home-timeline` microservice from DeathStarBench (GAN et al., 2019)'s social network application and introduce the NetCache accelerator between the timeline service and client storage service. In particular, the switch replies instead of the client storage service when the timeline service makes a request due to our modifications. We inject two problems: one of our INC problems (e.g., misconfiguration problem) and an x86 specific problem (we call this a `POST` problem). For the `POST` problem, we disrupt the post storage service, which is invoked right after the client lookup, in order to retrieve the posts that should show on a user's `home-timeline`.

From Fig. 4.6b, we observe that, as expected, Mimir detects the INC specific problems (blue circles) and fails completely to detect errors at the x86 (orange X), while *AppTracing* only detects the x86 specific problem with complete accuracy (purple diamonds) and suffers to detect the configuration problem (orange squares). Most importantly, we observe that we identify and effectively localize both issues combined in an end-to-end manner (`E2E`).

Figure 4.6: Simulated Localization Accuracy



Source: The Authors (2022)

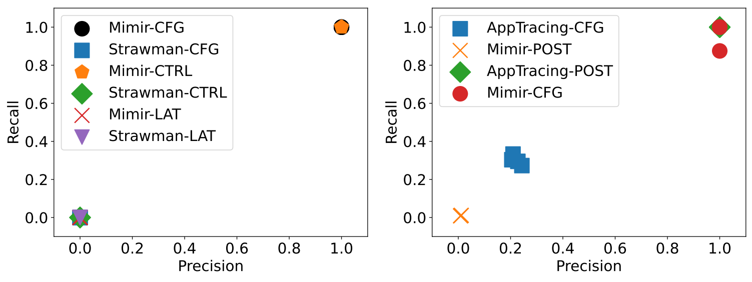
4.5.4 Overheads and Scalability

We evaluate the overheads and scalability of Mimir on our pyhsical testbed with a Tofino-based hardware switch.

Overheads. We focus on the switch ASIC resource overheads, in particular concentrating on sALU and memory. Table 4.6 shows the resource usage of different INCs with (`Mimir`) and without (`Base`) Mimir instrumentation. We see that our SRAM overheads are always marginal due to the reduced size of our histograms. Our TCAM usage (from the log_2 approximation algorithm) is bound by the timestamp bit-width and thus will also be negligible (sub 100 entries). Our SALU impact is a function of the number of annotated RPCs, which are low enough that we did not need to implement any optimizations such as register-sharing for our histograms. Finally, we conducted experiments

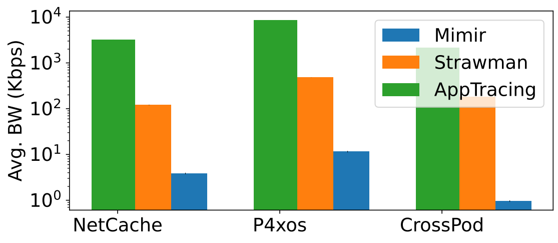
Figure 4.7: Localization Accuracy

(a) Tofino Localization Accuracy (b) E2E Localization Accuracy



Source: The Authors (2022)

Figure 4.8: Emulated Trace Export Overhead



Source: The Authors (2022)

on the throughput of register polling and found that a complete polling period for Mimir registers on our NetCache prototype takes approximately 13us.

Scalability. Now, we turn our attention to understanding the scalability of Mimir with respect to the bandwidth overheads associated with exporting data from a network of switches to a central entity (Figure not shown due to space). Abstractly, we observe that for Mimir the rate of requests has minimal impact on network bandwidth because Mimir exports constant sized data aggregates and alert information. We note, however (not shown due to space), that total network bandwidth is naturally a linear function of topology because each switch needs to generate reports. The alerts generated by Mimir are also a function of the number and type of problems. Conversely, both the topology and request rate will cause *Strawman* to have higher overhead.

To understand the broader deployment context, and put Mimir's overheads in perspective, we provide export overheads in Fig. 4.8 for the different INCs. Looking across these INCs, we observe variance in bandwidth overheads and identify that this variance is due to the number of INCs deployed in the network. More importantly, we observe

Table 4.6: Tofino Pipeline Resource Occupancy

Resource	NetCache		P4xos		CrossPod	
	Base	Mimir	Base	Mimir	Base	Mimir
sRAM	36.3%	37.3%	44.0%	44.7%	3.6%	4.7%
sALU	66.7%	72.9%	60.4%	64.6%	25.0%	31.3%
TCAM	5.6%	5.9%	0.3%	0.7%	0.3%	1.0%

Source: The Authors (2022)

that Mimir's overheads, even at low RPS, are orders of magnitude lower compared to competing techniques. In particular, adding Mimir to *AppTracing* to provide complete End-to-End visibility over infrastructures with offloaded components would only provide a relatively small increase in bandwidth overheads.

4.5.5 Understanding Mimir

We conclude by examining the components within Mimir to understand how they contribute to its scalability and accuracy. To understand different design points, we increment *Strawman* (Unoptimized Foxhound) with two orthogonal techniques from Mimir. Firstly, we make use of packet headers to coordinate span information into *Strawman* to create *Strawman++*. This is a form of mix-and-match between the techniques of Foxhound and Mimir that was hinted on earlier in the introduction chapter and helps us understand the performance impact of each optimization.

Reducing the Tracing Data Stream. We begin by comparing our use of a decoupled storage layer for aggregation against *Strawman*, which generates and exports spans (Figure not shown due to space). We observe that *Strawman* can export approximately 400k spans per second whereas temporarily storing information in the header (*i.e.*, *Strawman++*) and exploring all information together in one span increases the export rate to approximately 750k spans per second. Recall, *Strawman* exports spans for every traced packet which enters the switch (*i.e.*, once for NetCache read-request, another for read-reply). In both cases, performance is limited by known PCI-e problems (NEUGEBAUER et al., 2018) (e.g., hardware constraints and software inefficiencies). Regardless, at higher RPS only Mimir maintains high coverage due to in-network aggregation.

4.6 Discussion & Limitations

Debugging: Our approach extends and supports a sufficient set of tracing primitives and annotations to detect and localize problems. However, problem resolution requires debugging the root cause either by examining snapshots of variables or developer log files. As part of future work, we plan to extend Mimir to support richer annotation to enable logs and variable snapshots.

Targeted Diagnosis: While we focus on diagnosing the entire infrastructure, operators are often interested in focusing their efforts on specific applications/tenants or on workflows that include specific services. We are extending Mimir to address these scenarios by translating the "scope" into tags that the agent adds. Given these tags, Mimir maintains an additional data structure to store the focused aggregates. We are exploring existing extensions to P4 (SONCHACK et al., 2021) to instantiate and effectively maintain these additional data structures dynamically.

Random Sampling: Currently, Mimir only generates spans based on request-level performance procedures. This is traditionally known as tail-based sampling. In truth, most operators also use random sampling to generally understand their infrastructures. While Mimir does not generate random spans, we note that our aggregates can be combined with the aggregated random sampling to understand trends.

5 RELATED WORK

In this section we review the state-of-the-art in network telemetry (Section 5.1. and Section 5.2), distributed tracing (Section 5.3), and finally general debugging of network programs (Section 5.4). We focus our discussions on the works that more closely relate to our two systems.

5.1 Telemetry data exported into packet headers

Raw telemetry data (e.g., observed queue size) can be exported into packet headers (BASAT et al., 2020b; JEYAKUMAR et al., 2014b; KIM et al., 2015) as network traffic goes through switches (previously discussed in Chapter 2). Collected information is sent to a centralized entity for post-processing and for mining relevant statistics. Overall, the in-header approach demands high bandwidth and storage overheads per packet. Further, this technique imposes high CPU overheads due to the need for processing each individual packet header.

INT (KIM et al., 2015). In-band Network Telemetry (INT) has become a well-established technique for PDP telemetry due to several reasons such as the high level of visibility it provides and the ease of implementation on current programmable hardware. In the context of INCs, despite allowing for extremely fine-grained telemetry, INT fundamentally lacks coordination between INT sources (switches that insert INT headers) and x86 servers and is therefore unsuited for diagnosis of cross-plane executions. Additionally, because INT is stored on packet headers that can be created at every hop, collecting a large amount of metadata would quickly and significantly increase the bandwidth overhead of INT. Recent efforts such as Probabilistic-INT (PINT (BASAT et al., 2020b)) have been successful in mitigating some of the INT overhead by trading-off accuracy and bandwidth.

Kodeswaran et al. (KODESWARAN et al., 2020) does code path profiling of P4 programs. They implement the efficient Ball-Larus algorithm for profiling the path taken by individual packets. A noteworthy difference is that they do so without programmer input (*e.g.*, annotations), purely focusing on efficiently capturing a packet's path (*e.g.,* traversed tables and functions) within a program's control flow graph. This path profiling closely relates to Mimir's structural profiling. However, *Kodeswaran et al.* profiles the structure of source code while Mimir profiles the structure of trace DAGs. Additionally, Mimir aggregates the DAG structures directly in switch counters as distributions, while *Kodeswaran et al.* does not aggregate their optimized bitmask, which can instead be put into each packet's header for INT-like postprocessing.

5.2 Telemetry data aggregated by switches

Network switches can be used as memory buffers to aggregate (e.g., per-flow packet count) and export records (GUPTA et al., 2018; NARAYANA et al., 2017a; SONCHACK et al., 2018). This class of works essentially uses a combination of retrospection and packet mirroring (Chapter 2). However, this can create potentially high bandwidth and CPU demands: even aggregation might not be enough, for example, when the number of flows becomes too large. Alternatively, instead of exporting per-flow records, a switch can aggregate flow data into fixed-size sketches (*e.g.*, Nitrosketch (LIU et al., 2019b), UnivMon (LIU et al., 2016)). Both this and the above forms of switch aggregation provide a much lighter exporting scheme compared to using packet headers. Also, in-network aggregation into sketches can save even more bandwidth when exporting data, i.e., by aggregating records into statistical data structures. Foxhound piggybacks on this idea by allowing for customized sketch structures to be deployed for its queries. Likewise, Mimir employs in-network data structures to minimize overheads, leveraging data structures built around the procedures which they offload.

Marple (NARAYANA et al., 2017a). Marple is a query-driven telemetry system which supports only queries that can be aggregated at the switch ASIC. While Marple allows scalable queries over headers of packet-ins and outs (network telemetry), Mimir and Foxhound focus on providing abstractions for capturing the computation in-between (in-network compute). Despite that difference, their storage layers share insights. Specifically, Marple instantiates a hashtable at the switch to store key-value pairs and opportunistically aggregate query data early (akin to Mimir and some of the Foxhound optimizations demonstrated earlier).

Sonata (GUPTA et al., 2018). Sonata is a query-oriented telemetry framework which employs both retrospection for data that can be efficiently aggregated in-switch (akin to Marple), and packet mirroring otherwise. The latter case has Sonata mirror packets to a stream processor. This strategy incurs in high overhead, however, Sonata manages to mitigate some of this overhead by restricting the conditions for exporting packets (*e.g.,* only export when a packet is destined for port 22). This filtering generally happens inside the Sonata-powered switch and the conditions for filtering are extracted automatically from Sonata queries. With respect to *aggregated* data, Sonata deploys a hash-table for its queries.

SpiderMon (WANG et al., 2020b). Spidermon focuses on finding performance degradation root causes using historical data about switch queue occupancy. Like Mimir, Spidermon decomposes an end-to-end investigation into device-local ones. When drops are detected, SpiderMon audits switch "logs" in order to discover who caused the problem. SpiderMon, as well as our two systems, argue for the use of upfront processing at ASICs, based on the data we need for analysis. However, while Foxhound and Mimir focus on data and procedures for localizing general in-network compute problems, SpiderMon optimizes a well-known problem (network contention) and tries to perform root cause analysis for that specific problem. In other words, while SpiderMon is optimized to diagnose network performance degradation, the arbitrary information relevant to understanding performance degradation of INCs cannot be contemplated with the general metrics and analysis proposed.

5.3 x86 Distributed Tracing

This corresponds to traditional distributed tracing in which a tracing agent collects information from traces of a x86 application (e.g., Jaeger (JAEGER, 2020), Zipkin (ZIPKIN, 2021)). Although distributed tracing has been for many years the *de-facto* standard for observability, it does not account for what is happening within the in-network computing. As such, distributed tracing can detect problems from the application level but cannot see into the network and thus fails to localize problematic switches involved in an RPC call.

SpectroScope (SAMBASIVAN et al., 2011). SpectroScope is a system that relies heavily on aggregating x86 distributed traces and performing troubleshooting on the aggregates. Mimir is closely related to SpectroScope as both works attempt to troubleshoot the aggregates and not the raw traces. However, SpectroScope operates with the traditional separation between data collection and processing. Basically, SpectroScope will aggregate traces from a database filled with traces which have already been collected. Spectroscope will then diagnose the aggregates. Given the additional constraints of PDPs, having a database filled with raw traces is cumbersome and may result in heavy sampling, and thus Mimir optimizes data collection to collect only the aggregates necessary for postprocessing and diagnosis.

Fay (ERLINGSSON et al., 2012). Fay is a tracing platform that allows the specification of queries that define both what to trace via dynamic instrumentation and early aggregation, as well as how to process and combine trace events from multiple sources (data-parallel computation). Fay focuses these ideas to collect metrics based on local (not distributed) traces of kernel functions. Fay shares high level ideias with Mimir about aggregation and early trimming of the data flow. Essentially, Fay offloads metrics aggregation to the devices producing the metrics, allowing them to aggregate early and reduce data footprint. Conversely, Mimir argues for decomposing certain end-to-end distributed tracing workflows (*e.g.*, structural and temporal aggregation) down to the device-level, where we can also aggregate early.

5.4 Other Debugging Approaches

This section overviews a few alternative techniques which have been explored to troubleshoot problems in programmable data planes. Instead of statically instrumenting run-time data collection – a standard practice of observability frameworks – these techniques explore alternative directions.

Simulation. Clara (QIU et al., 2020) presents a simulation-based system for predicting the behaviour of P4 programs when deployed onto different SmartNIC architectures. This prediction tackles hard-to-simulate measurements (such as processing latency) by modelling specific information from each SmartNIC. Differently from Foxhound and Mimir, Clara does not aim at capturing actual runtime measurements, but rather simulate them to provide performance insights.

Verification. Works like p4v (LIU et al., 2018) allow programmers to use annotations to verify P4 programs. It translates P4 programs to Guarded Command Language (GCL) and then uses a theorem prover for checking properties of interest. Similarly to p4v, Assert-P4 (FREIRE et al., 2018) also uses annotations directly in P4 source code to define invariant conditions written as assertions. It then translates P4 code to C and symbolically executes this code. Bugs are discovered if at any point the code reaches an assertion and the invariant is violated. Vera (STOENESCU et al., 2018) also translates P4 code and performs a symbolic execution of the program. However, Vera is able to identify a range of bugs automatically without assertions. Vera also translates P4 code to an optimized language (STOENESCU et al., 2016). While most works aimed at debugging P4 programs attempt to debug and guarantee properties about switch code alone, our work encompasses a more comprehensive view of the network, tracing executions from and to servers and their distributed applications.

Dynamic Instrumentation. PhD (SULTANA et al., 2017) is a query-oriented FPGA debugging system. PhD dynamically instruments FPGAs to execute remote code from user queries to control and observe the flow of a program. While this high level of flexibility favours debugging purposes, the intended use of our systems for in-network tracing in production environments prohibits us to. Furthermore, switching ASICs such as the Barefoot Tofino do not support the proposed dynamic re-instrumentation that PhD leverages. The absence of sketch structures and overall aggregation also makes PhD queries unable to scale more demanding queries to line rate.

6 CONCLUSION

6.1 Summary of Contributions

Our work began with the motivation of bridging the gap between INC functionality and diagnosis. In developing our solution, we identified that there was a clear trade-off between flexibility and scalability. We decided then to treat each of these dimensions as a different problem. Ultimately, this work presents two steps towards in-network tracing: one towards expressiveness (Foxhound) and the other towards scalability (Mimir).

Foxhound abridges the entire spectrum of telemetry, from *isolate* to *aggregate* data and allows operators to write expressive queries and also optimize them. Foxhound is a proof-of-concept that PDP limitations can be bypassed to enable flexible diagnosis (as seen by our many queries and our evaluation).

Mimir, on the other hand, focuses on a subset of tracing queries and optimizes them to be implemented in hardware and scale out to more demanding workloads. Mimir is a proof-of-concept a handful of “catch-all” queries amenable to aggregation (e.g., profiling DAG structure distributions) can be effective in diagnosing common INC problems. Moreover, Mimir’s optimizations showcase the benefits of offloading part of these queries into the data plane and (differently from traditional tracing) the benefits of designing a storage layer around these queries, promoting the early thinning of the dataflow and thus alleviating several bottlenecks (including the PCI bottleneck which plagued Foxhound).

We also note that an important overlap of Foxhound and Mimir is that, for certain queries, the practical export limitations of current programmable switches can be mitigated by using domain-specific knowledge about a given tracing query (e.g., what traces do we care about?) to inform data collection. Due to practical concerns, this kind of optimization remains unexplored by most x86 distributed tracing libraries in the wild, but we argue that it is an important thread of research.

While Mimir and Foxhound stand on widely different points in the design space, our decomposition of both systems into their more fundamental pieces allows us to judge each optimization for what it is worth and create an entire spectrum of possible mixing and matching between Foxhound and Mimir techniques (with even the possibility of inter-operating queries from both systems in the same switch). This alleviates the main limitation of our work: the fact that our main optimizations are query-specific. Specifically, there can be situations (queries) in which Mimir’s disaggregated tracing may lose important end-to-end information. In this case, a Foxhound query may be more adequate as it can provide end-to-end traces. Alternatively, Foxhound queries may suffer from sampling if the workloads are too demanding or if the query relies on having a high-fidelity set of traces. Our work presents techniques and optimizations that allow users to maximize one dimension or the other, but even so, we do not offer a complete solution for efficiently implementing all possible queries.

Finally, it is our sincere hope that the motivation and "lessons learned" presented in this work (especially those related to scalability) as well as the underlying body of techniques and optimizations that power Foxhound and Mimir can serve to strengthen the state-of-the-art in INC observability and reduce the barrier of adoption for actual INC-accelerated environments.

6.2 Future Work

Many threads emerged during this work that could be pursued as future work such as:

- **Tuning PCI-e communication protocols to increase tracing throughput:** As we saw in Section 3.8.4.1, the PCIe interface is actually an important bottleneck for low-overhead tracing efforts. While this interface has evolved in programmable switches (when compared to fixed-function ones, (SONCHACK et al., 2018)), more low-level work could be done to optimize the usage of the PCIe interface, for example, with a batching primitive which allows PDP spans to be continuously accumulated into a large ethernet frame which, once full, would be exported. This would remove the high resource pressure that batching has on the programmable pipeline (Section 3.8.4.1), and instead implements batching as a hardware primitive of Tofino.

- **Incorporating logs and metrics into our PDP observability framework:** DevOps are accustomed to the wide range of observability techniques needed to debug server applications. Troubleshooting in large distributed systems generally follows a pipeline, iterating and drilling down from high-level performance indicators of problems (*what?* – global metrics and performance indicators), followed by localization of the offending component inside the infrastructure (*where?* – distributed tracing) and finally debugging the problem with machine-centric retrospection to the specific code paths (*which?* – logs). For our systems to be complete, we would need to include some notion of those three data sources (metrics, traces, and logs).

- **More Real-World INC data and Root Cause Analysis:** Our designs themselves could be more thoroughly evaluated if real-world INC data were publicly available. We believe that this validation is crucial and takes precedence over other kinds of future work. Additionally, such datasets would open up the possibility of mining and identifying the signatures of certain classes of bugs. This identification of bug signatures would allow our systems to attempt root cause analysis. For example, a control failure on NetCache has a unique signature in the structure of the traced DAGs: after the failure, no outgoing edges come out of the switch CPU node. While this is an over-simplification of a real-world signature, more robust RCA techniques exist and have been demonstrated to be feasible (WANG et al., 2020b).

- **SmartNIC Observability:** SmartNICs have also been shown to be able to offload computation (LIU et al., 2019a). Thus, we reason that the motivation behind Foxhound and Mimir would also apply to SmartNICs which perform in-network computation. While this work focused solely on programmable switches (e.g., Intel Tofino), we plan to study the SmartNIC environment in order to discover how well the principles of our work would apply to SmartNICs. In that regard, general-purpose SmartNICs offer a higher degree of flexibility when compared to some of the limitation of P4 programmable data planes.

REFERENCES

ALIBABA. **Alibaba Tracing-Analysis**. 2021. <https://www.alibabacloud.com/product/tracing-analysis>.

ANAND, V. et al. Aggregate-driven trace visualizations for performance debugging. **CoRR**, abs/2010.13681, 2020. Available from Internet: <https://arxiv.org/abs/2010.13681>.

BASAT, R. B. et al. Pint: Probabilistic in-band network telemetry. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM ’20), p. 662–680. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405894>.

BASAT, R. B. et al. Pint: Probabilistic in-band network telemetry. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM ’20), p. 662–680. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405894>.

BENSON, T. A. In-network compute: Considered armed and dangerous. In: . [S.l.: s.n.], 2019. (HotOS ’19).

BERG, J. et al. Snicket: Query-driven distributed tracing. In: **Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2021. (HotNets ’21), p. 206–212. ISBN 9781450390873. Available from Internet: <https://doi.org/10.1145/3484266.3487393>.

BMV2 Performance. <https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md>. 2022.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656890>.

CHEN, X. et al. Beaucoup: Answering many network traffic queries, one memory update at a time. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM ’20), p. 226–239. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405865>.

CHOW, M. et al. The mystery machine: End-to-end performance analysis of large-scale internet services. In: **OSDI ’14**. Broomfield, CO: [s.n.], 2014.

CISCO. **Introduction to Cisco IOS NetFlow - A Technical Overview**. 2019. <https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html>. Last accessed in Oct 9th, 2019.

CORMODE, G.; HADJIELEFTHERIOU, M. Methods for finding frequent items in data streams. **The VLDB Journal**, Springer-Verlag, Berlin, Heidelberg, v. 19, n. 1, p. 3–20, feb 2010. ISSN 1066-8888. Available from Internet: <https://doi.org/10.1007/s00778-009-0172-z>.

DANG, H. T. et al. P4xos: Consensus as a network service. **IEEE/ACM Trans. Netw.**, IEEE Press, v. 28, n. 4, p. 1726–1738, aug 2020. ISSN 1063-6692. Available from Internet: <https://doi.org/10.1109/TNET.2020.2992106>.

DANG, H. T. et al. Whippersnapper: A p4 language benchmark suite. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSR '17), p. 95–101. ISBN 9781450349475. Available from Internet: <https://doi.org/10.1145/3050220.3050231>.

DOCKER compose. <https://docs.docker.com/compose/>.

DUMITRESCU, D. et al. Bf4: Towards bug-free p4 programs. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 571–585. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405888>.

ERLINGSSON, U. et al. Fay: Extensible distributed tracing from kernels to clusters. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 30, n. 4, nov 2012. ISSN 0734-2071. Available from Internet: <https://doi.org/10.1145/2382553.2382555>.

FONSECA, R. et al. X-trace: A pervasive network tracing framework. In: **4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)**. Cambridge, MA: USENIX Association, 2007. Available from Internet: <https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework>.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: . ACM, 2018. (SOSR '18), p. 4:1–4:7. ISBN 978-1-4503-5664-0. Available from Internet: <http://doi.acm.org/10.1145/3185467.3185499>.

GAN, Y. et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: **Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2019. (ASPLOS '19), p. 3–18. ISBN 9781450362405. Available from Internet: <https://doi.org/10.1145/3297858.3304013>.

GOOGLE. **Google Cloud Trace**. 2021. <https://cloudplatform.googleblog.com/2015/01/Diagnose-Service-Performance-Bottlenecks-with-Google-Cloud-Trace-Beta.html>.

GRAFANA. **Grafana Tail Sampling**. 2020. <https://grafana.com/docs/tempo/latest/grafana-agent/tail-based-sampling/>.

GUO, X. et al. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1387–1397. ISBN 9781450370431. Available from Internet: <https://doi.org/10.1145/3368089.3417066>.

GUO, X. et al. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1387–1397. ISBN 9781450370431. Available from Internet: <https://doi.org/10.1145/3368089.3417066>.

GUPTA, A. et al. Sonata: Query-driven streaming network telemetry. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 357–371. ISBN 9781450355674. Available from Internet: <https://doi.org/10.1145/3230543.3230555>.

HANDIGOL, N. et al. 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)**. Seattle, WA: USENIX Association, 2014. p. 71–85. ISBN 978-1-931971-09-6. Available from Internet: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol>.

HUANG, L.; ZHU, T. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In: CURINO, C.; KOUTRIKA, G.; NETRAVALI, R. (Ed.). **SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021**. ACM, 2021. p. 76–91. Available from Internet: <https://doi.org/10.1145/3472883.3486994>.

INT Use Case. <https://www.barefootnetworks.com/use-cases/>. Last accessed in Oct 9th, 2019.

INTEL. **Intel Tofino**. 2022.

JAEGER. **Jaeger open source, end-to-end distributed tracing**. 2020. <https://www.jaegertracing.io>.

JEPSEN, T. et al. Fast string searching on pisa. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2019. (SOSR '19), p. 21–28. ISBN 9781450367103. Available from Internet: <https://doi.org/10.1145/3314148.3314356>.

JEYAKUMAR, V. et al. Millions of little minions: Using packets for low latency network programming and visibility. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 4, p. 3–14, aug. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2740070.2626292>.

JEYAKUMAR, V. et al. Millions of little minions: Using packets for low latency network programming and visibility. In: **Proceedings of the 2014 ACM Conference on SIGCOMM**. New York, NY, USA: Association for Computing Machinery, 2014. (SIGCOMM '14), p. 3–14. ISBN 9781450328364. Available from Internet: <https://doi.org/10.1145/2619239.2626292>.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: . [S.l.: s.n.], 2017. (SOSP '17).

JIN, X. et al. Netchain: Scale-free sub-rtt coordination. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. Renton, WA: USENIX Association, 2018. p. 35–49. ISBN 978-1-939133-01-4. Available from Internet: <https://www.usenix.org/conference/nsdi18/presentation/jin>.

JOSE, L. et al. Compiling packet programs to reconfigurable switches. In: **Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2015. (NSDI'15), p. 103–115. ISBN 9781931971218.

JOUKOV, N. et al. Operating system profiling via latency analysis. In: **Proceedings of the 7th Symposium on Operating Systems Design and Implementation**. USA: USENIX Association, 2006. (OSDI '06), p. 89–102. ISBN 1931971471.

KALDOR, J. et al. Canopy: An end-to-end performance tracing and analysis system. In: . [S.l.: s.n.], 2017. (SOSP '17).

KAMON. **Kamon Telemetry**. 2022. <https://kamon.io/blog/how-to-keep-traces-for-slow-and-failed-requests>.

KANNAN, P. G. et al. Debugging transient faults in data centers using synchronized network-wide packet histories. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. USENIX Association, 2021. p. 253–268. ISBN 978-1-939133-21-2. Available from Internet: <https://www.usenix.org/conference/nsdi21/presentation/kannan>.

KANNAN, P. G.; JOSHI, R.; CHAN, M. C. Precise time-synchronization in the data-plane using programmable switching asics. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. New York, NY, USA: ACM, 2019. (SOSR '19), p. 8–20. ISBN 978-1-4503-6710-3. Available from Internet: <http://doi.acm.org/10.1145/3314148.3314353>.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: . [S.l.: s.n.], 2015.

KODESWARAN, S. et al. Tracking p4 program execution in the data plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2020. (SOSR '20), p. 117–122. ISBN 9781450371018. Available from Internet: <https://doi.org/10.1145/3373360.3380843>.

LAMPORT, L. The part-time parliament. **ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].**, May 1998. ACM SIGOPS Hall of Fame Award in

2012. Available from Internet: <https://www.microsoft.com/en-us/research/publication/part-time-parliament/>.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In: **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2010. (Hotnets-IX). ISBN 9781450304092. Available from Internet: <https://doi.org/10.1145/1868447.1868466>.

LAO, C. et al. ATP: In-network aggregation for multi-tenant learning. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. USENIX Association, 2021. p. 741–761. ISBN 978-1-939133-21-2. Available from Internet: <https://www.usenix.org/conference/nsdi21/presentation/lao>.

LI, J. et al. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In: **14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)**. USENIX Association, 2020. p. 387–406. ISBN 978-1-939133-19-9. Available from Internet: <https://www.usenix.org/conference/osdi20/presentation/li-jialin>.

LI, Y. et al. DETER: Deterministic TCP replay for performance diagnosis. In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. Boston, MA: USENIX Association, 2019. p. 437–452. ISBN 978-1-931971-49-2. Available from Internet: <https://www.usenix.org/conference/nsdi19/presentation/li-yuliang>.

LIGHTSTEP. **Lightstep**. 2022. <https://lightstep.com>.

LIU, H. et al. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In: SILVA, D. D.; WANG, Q.; ZHANG, L.-J. (Ed.). **Cloud Computing – CLOUD 2019**. Cham: Springer International Publishing, 2019. p. 287–302. ISBN 978-3-030-23502-4.

LIU, J. et al. P4v: Practical verification for programmable data planes. In: . [S.l.: s.n.], 2018. (SIGCOMM '18).

LIU, M. et al. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In: **2019 USENIX Annual Technical Conference (USENIX ATC 19)**. Renton, WA: USENIX Association, 2019. p. 363–378. ISBN 978-1-939133-03-8. Available from Internet: <https://www.usenix.org/conference/atc19/presentation/liu-ming>.

LIU, Z. et al. Nitrosketch: Robust and general sketch-based monitoring in software switches. In: **Proceedings of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 334–350. ISBN 9781450359566. Available from Internet: <https://doi.org/10.1145/3341302.3342076>.

LIU, Z. et al. One sketch to rule them all: Rethinking network flow monitoring with univmon. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 101–114. ISBN 9781450341936. Available from Internet: <https://doi.org/10.1145/2934872.2934906>.

MACE, J. **End-to-End Tracing: Adoption and Use Cases**. [S.l.], 2017.

MACE, J.; FONSECA, R. Universal context propagation for distributed system instrumentation. In: . [S.l.: s.n.], 2018. (EuroSys '18).

MACE, J.; ROELKE, R.; FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: **Proceedings of the 25th Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2015. (SOSP '15), p. 378–393. ISBN 978-1-4503-3834-9. Available from Internet: <http://doi.acm.org/10.1145/2815400.2815415>.

MASSEY, F. J. The kolmogorov-smirnov test for goodness of fit. **Journal of the American Statistical Association**, [American Statistical Association, Taylor Francis, Ltd.], v. 46, n. 253, p. 68–78, 1951. ISSN 01621459. Available from Internet: <http://www.jstor.org/stable/2280095>.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014.

NARAYANA, S. et al. Language-directed hardware design for network performance monitoring. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 85–98. ISBN 9781450346535. Available from Internet: <https://doi.org/10.1145/3098822.3098829>.

NARAYANA, S. et al. Language-directed hardware design for network performance monitoring. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 85–98. ISBN 9781450346535. Available from Internet: <https://doi.org/10.1145/3098822.3098829>.

NETFLIX. **Netflix's Inca Autotracing System**. 2022. <https://www.infoq.com/presentations/netflix-streaming-data-infrastructure/?utm_source=youtube&utm_medium=link&utm_campaign=qcontalks>.

NEUGEBAUER, R. et al. Understanding pcie performance for end host networking. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 327–341. ISBN 9781450355674. Available from Internet: <https://doi.org/10.1145/3230543.3230560>.

OPENTELEMETRY. 2021. <https://opentelemetry.io>.

P4LANG. **Behavioral Model**. 2020. <https://github.com/p4lang/behavioral-model>.

PANARETOS, V. M.; ZEMEL, Y. Statistical aspects of wasserstein distances. **Annual Review of Statistics and Its Application**, Annual Reviews, v. 6, n. 1, p. 405–431, Mar 2019. ISSN 2326-831X. Available from Internet: <http://dx.doi.org/10.1146/annurev-statistics-030718-104938>.

PINTEREST. **PinTrace**. 2022. <https://medium.com/pinterest-engineering/distributed-tracing-at-pinterest-with-new-open-source-tools-a4f8a5562f6b>.

QIU, Y. et al. Clara: Performance clarity for smartnic offloading. In: **Proceedings of the 19th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2020. (HotNets '20), p. 16–22. ISBN 9781450381451. Available from Internet: <https://doi.org/10.1145/3422604.3425929>.

SAMBASIVAN, R. R. et al. Principled workflow-centric tracing of distributed systems. In: **Proceedings of the Seventh ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2016. (SoCC '16), p. 401–414. ISBN 9781450345255. Available from Internet: <https://doi.org/10.1145/2987550.2987568>.

SAMBASIVAN, R. R. et al. Diagnosing performance changes by comparing request flows. In: **8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)**. Boston, MA: USENIX Association, 2011. Available from Internet: <https://www.usenix.org/conference/nsdi11/diagnosing-performance-changes-comparing-request-flows>.

SANVITO, D.; SIRACUSANO, G.; BIFULCO, R. Can the network be the ai accelerator? In: **Proceedings of the 2018 Morning Workshop on In-Network Computing**. New York, NY, USA: Association for Computing Machinery, 2018. (NetCompute '18), p. 20–25. ISBN 9781450359085. Available from Internet: <https://doi.org/10.1145/3229591.3229594>.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2017. (HotNets-XVI), p. 150–156. ISBN 978-1-4503-5569-8. Available from Internet: <http://doi.acm.org/10.1145/3152434.3152461>.

SHKURO, Y. Observability infra, uber and facebook. In: . San Jose: [s.n.], 2019. (Systems@Scale'19).

SIGELMAN, B. H. et al. **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**. [S.l.], 2010. Available from Internet: <https://research.google.com/archive/papers/dapper-2010-1.pdf>.

SIVARAMAN, A. et al. Packet transactions: High-level programming for line-rate switches. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 15–28. ISBN 9781450341936. Available from Internet: <https://doi.org/10.1145/2934872.2934900>.

SMIRNOV, N. Approximate distribution laws for random variables, constructed from empirical data. **Uspekhi Mat. Nauk**, n. 10, 1944.

SONCHACK, J. et al. Turboflow: Information rich flow record generation on commodity switches. In: . [S.l.: s.n.], 2018. (EuroSys '18).

SONCHACK, J. et al. Lucid: A language for control in the data plane. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCOMM '21), p. 731–747. ISBN 9781450383837. Available from Internet: <https://doi.org/10.1145/3452296.3472903>.

SONCHACK, J. et al. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In: **ATC '18**. [S.l.: s.n.], 2018.

STEPHENS, B. E. et al. Tcp is harmful to in-network computing: Designing a message transport protocol (mtp). In: **Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2021. (HotNets '21), p. 61–68. ISBN 9781450390873. Available from Internet: <https://doi.org/10.1145/3484266.3487382>.

STOENESCU, R. et al. Debugging p4 programs with vera. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 518–532. ISBN 978-1-4503-5567-4. Available from Internet: <http://doi.acm.org/10.1145/3230543.3230548>.

STOENESCU, R. et al. **SymNet: scalable symbolic execution for modern networks**. 2016.

SULTANA, N. et al. Extending programs with debug-related features, with application to hardware development. **CoRR**, abs/1705.09902, 2017. Available from Internet: <http://arxiv.org/abs/1705.09902>.

SULTANA, N. et al. Flightplan: Dataplane disaggregation and placement for p4 programs. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. USENIX Association, 2021. p. 571–592. ISBN 978-1-939133-21-2. Available from Internet: <https://www.usenix.org/conference/nsdi21/presentation/sultana>.

Theis, T. N.; Wong, H. . P. The end of moore's law: A new beginning for information technology. **Computing in Science Engineering**, v. 19, n. 2, p. 41–50, Mar 2017. ISSN 1558-366X.

TIAN, B. et al. Aquila: A practically usable verification system for production-scale programmable data planes. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCOMM '21), p. 17–32. ISBN 9781450383837. Available from Internet: <https://doi.org/10.1145/3452296.3472937>.

TIRMAZI, M. et al. Cheetah: Accelerating database queries with switch pruning. In: . [S.l.: s.n.], 2019. (SIGCOMM Posters and Demos '19). ISBN 9781450368865.

TIRMAZI, M. et al. Cheetah: Accelerating database queries with switch pruning. In: **Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM Posters and Demos '19), p. 72–74. ISBN 9781450368865. Available from Internet: <https://doi.org/10.1145/3342280.3342311>.

TRAEGER, A. **Analyzing Root Causes of Latency Distributions**. Thesis (PhD), 2008.

TRAEGER, A.; DERAS, I.; ZADOK, E. DARC: dynamic analysis of root causes of latency distributions. In: LIU, Z.; MISRA, V.; SHENOY, P. J. (Ed.). **Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008**. ACM, 2008. p. 277–288. Available from Internet: <https://doi.org/10.1145/1375457.1375489>.

VARDI, M. Y. Moore's law and the sand-heap paradox. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 57, n. 5, p. 5, may 2014. ISSN 0001-0782. Available from Internet: <https://doi.org/10.1145/2600347>.

VASERSTEIN, L. N. Markov processes over denumerable products of spaces, describing large systems of automata. v. 5, n. 3, p. 64–72, 1969.

WANG, S. et al. Martini: Bridging the gap between network measurement and control using switching asics. In: **2020 IEEE 28th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2020. p. 1–12.

WANG, W. et al. Grasp the root causes in the data plane: Diagnosing latency problems with spidermon. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2020. (SOSR '20), p. 55–61. ISBN 9781450371018. Available from Internet: <https://doi.org/10.1145/3373360.3380835>.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets '19), p. 25–33. ISBN 9781450370202. Available from Internet: <https://doi.org/10.1145/3365609.3365864>.

YAKIMOV, E. Tracing real-time distributed systems. In: . Dublin: USENIX Association, 2019. (SREcon 19 Europe/Middle East/Africa).

YU, D. et al. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. [S.l.: s.n.], 2019.

YU, Z. et al. Netlock: Fast, centralized lock management using programmable switches. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 126–138. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405857>.

ZHANG, Y. et al. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCOMM '21), p. 207–222. ISBN 9781450383837. Available from Internet: <https://doi.org/10.1145/3452296.3472892>.

ZHENG, P.; BENSON, T.; HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In: . [S.l.: s.n.], 2018. (CoNEXT '18).

ZHU, H. et al. Racksched: A microsecond-scale scheduler for rack-scale computers. In: **14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)**. USENIX Association, 2020. p. 1225–1240. ISBN 978-1-939133-19-9. Available from Internet: <https://www.usenix.org/conference/osdi20/presentation/zhu>.

ZIPKIN. **Zipkin**. 2021. <https://zipkin.io>.

APPENDIX A — RESUMO EXPANDIDO

Há um movimento crescente para descarregar a funcionalidade, por exemplo, TCP ou armazenamentos de valores-chave, na rede - em SmartNICs ou planos de dados programáveis. Embora o descarregamento prometa aumentos significativos de desempenho, esses dispositivos programáveis geralmente fornecem pouca visibilidade de seu desempenho. Além disso, muitas ferramentas existentes para analisar e depurar problemas de desempenho, por exemplo, rastreamento distribuído, não se estendem a esses dispositivos.

Motivado por essa falta de visibilidade, a primeira metade deste trabalho apresenta o design e implementação do Foxhound, um *framework* de observabilidade para computação em rede. Esse *framework* apresenta uma linguagem de consulta, um compilador e uma camada de abstração de armazenamento coprojetados para expressar, capturar e analisar rastreamentos distribuídos e seus dados de desempenho em uma infraestrutura que inclui servidores e planos de dados programáveis.

Embora o Foxhound seja nossa prova de conceito para rastreamento flexível na rede, descobrimos que o paradigma de rastreamento tradicional que o Foxhound incorpora pode sofrer de problemas de escalabilidade devido às limitações de hardware dos planos de dados programáveis. Em nosso esforço para mitigar isso, identificamos um subconjunto de consultas de rastreamento comuns que podem ser hiper-otimizadas mesmo além das otimizações do Foxhound. Essas otimizações representam um afastamento do rastreamento tradicional e constituem outro framework, o Mimir, apresentado na segunda metade deste trabalho. O Mimir troca a flexibilidade pela eficiência, explorando um conjunto de opções de design que otimizam tarefas comuns de diagnóstico e localização. Nossas avaliações usando três aplicativos descarregados representativos em um testbed baseado em Intel Tofino, um emulador e um simulador mostram que o Mimir pode suportar um subconjunto de tarefas de rastreamento comuns em escala com *overhead* significativamente menor do que o Foxhound. Além disso, nossos experimentos com um microsserviço do DeathStarBench aprimorado por computação em rede demonstram a utilidade de nossa abordagem para diagnóstico fim-a-fim.

Resumidamente, nosso trabalho como um todo tem a motivação de preencher a lacuna entre a funcionalidade proporcionada pelas INCs e o diagnóstico das INCs. Ao desenvolver nossas soluções, identificamos que havia um claro *trade-off* entre flexibilidade e escalabilidade. Decidimos então tratar cada uma dessas dimensões como um problema diferente. Em última análise, este trabalho apresenta dois passos para o rastreamento em rede: um para a expressividade (Foxhound) e outro para a escalabilidade (Mimir).

Foxhound conecta todo o espectro de telemetria, tanto dados isolados quanto dados agregados, e permite que os operadores escrevam consultas expressivas e também as otimizem. Foxhound é uma prova de conceito de que as limitações dos planos de dados programáveis podem ser contornadas para permitir um diagnóstico flexível (como visto por nossas muitas consultas e avaliações).

Mimir, por outro lado, concentra-se em um subconjunto de consultas de rastreamento e as otimiza para serem implementadas em hardware e dimensionadas para cargas de trabalho mais exigentes. Mimir é uma prova de conceito de que algumas consultas importantes e genéricas que são passíveis de agregação (por exemplo, perfis de distribuições de estrutura de *traces*) podem ser eficazes no diagnóstico de problemas comuns de INC. Além disso, as otimizações do Mimir mostram os benefícios de descarregar parte dessas consultas no plano de dados e (diferentemente do rastreamento tradicional) os benefícios de projetar uma camada de armazenamento em torno dessas consultas, promovendo a filtragem precoce do fluxo de dados e, assim, aliviando vários gargalos (incluindo o gargalo PCI que atormentou o Foxhound).

Enquanto Mimir e Foxhound estão em pontos amplamente diferentes no espaço de *design*, nossa decomposição de ambos os sistemas em suas peças mais fundamentais nos permite julgar cada otimização individualmente e criar um espectro inteiro de possíveis misturas e combinações entre as técnicas de Foxhound e Mimir (com até a possibilidade de interoperar consultas de ambos os sistemas no mesmo *switch*).

Finalmente, é nossa sincera esperança que a motivação e as “lições aprendidas” apresentadas neste trabalho (especialmente aquelas relacionadas à escalabilidade), bem como o corpo subjacente de técnicas e otimizações que constituem Foxhound e Mimir possam servir para fortalecer o estado da arte em observabilidade de INCs e reduzir a barreira de adoção para ambientes acelerados por INC.

Como principal trabalho futuro, prentendemos conseguir *datasets* reais de INCs. Pensamos que nosso trabalho poderia ser melhor avaliado se os dados INC do mundo real estivessem disponíveis publicamente. Acreditamos que esta validação é crucial e tem precedência sobre outros tipos de trabalhos futuros. Além disso, esses conjuntos de dados abririam a possibilidade de mineração e identificação das assinaturas de certas classes de *bugs*. Essa identificação de assinaturas de bugs permitiria que nossos sistemas tentassem a análise da causa raiz (*root cause analysis* ou RCA). Por exemplo, uma falha de controle no NetCache tem uma assinatura diferenciada na estrutura dos *DAGs* rastreados: após a falha, nenhuma aresta sai do nó da *CPU* do switch. Embora esta seja uma simplificação excessiva de uma assinatura do mundo real, existem técnicas de RCA mais robustas que demonstraram viabilidade e que poderiamos utilizar.