Language Expressiveness Under Extreme Scarcity in Programmable Data Planes

Mary Hogan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Jennifer Rexford

May 2024

Abstract

Today's networks must run a vast array of sophisticated applications that support services we rely on. These applications provide increased security (*e.g.,* firewalls), better performance (*e.g.,* congestion detection), and the ability to scale services (*e.g.,* caching). The network devices that implement these applications need to operate at high speeds (100+Gbps) and be flexible enough to adapt applications to changing requirements. Programmable network devices have emerged as a way to customize network functionality, while guaranteeing high-speed processing.

Writing applications for programmable devices, however, is notoriously difficult. Switches have a restrictive architecture to ensure line-rate processing, and their programming languages are very low-level. Programmers must understand how many resources (*e.g.,* memory) each piece of their program requires. Deploying these applications often requires tedious optimization of their layout, with programmers manually writing, compiling, and testing an implementation, adjusting the design, and repeating.

To better manage resource allocation, we present P4All, an extension of an existing programming language that allows programmers to define *elastic* data structures that stretch automatically to optimally use available resources. These structures are defined using *symbolic values* (that parameterize the size of the structure) and *objective functions* (that quantify the affect of size on performance). An optimization function specifies how to share resources amongst structures. We also create an optimizer that automatically finds the best resource allocation.

There are many other choices programmers make beyond resource allocation, some of which likely depend on the expected workload. To automate those decisions, we present Parasol, a framework that allows programmers to define general, parameterized applications and automatically optimize their parameters. The parameters can represent a variety of implementation decisions, and may be optimized for high-

level objectives defined by the programmer. Optimization is tailored to particular environments using a representative traffic trace.

We implement a diverse set of applications in P4All and Parasol to evaluate the optimizers, and we compile the resulting optimized programs to an actual programmable switch. P4All and Parasol decouple programming languages from data-plane hardware to lift the burden of reasoning about low-level details from programmers and make it easier to develop applications.

Acknowledgments

I am eternally grateful to my advisor, Jennifer Rexford, for her mentorship, patience, and encouragement throughout this process. Jen is truly the best advisor that I could've asked for. She taught me how to write a research paper, how to find problems that both challenge and excite me, and how to gracefully navigate professional hurdles and setbacks. She gave me the space and support to explore my passions and find my path, and I would not be the researcher I am today without her.

I would like to thank the members of my committee, David Walker, Shir Landau Feibish, Maria Apostolaki, and Ravi Netravali, for their helpful and insightful feedback, which has significantly improved this dissertation. Special thanks to Dave and Shir, who have been incredible collaborators and mentors throughout the duration of my PhD. They have helped me learn how to communicate my research and how to transform abstract ideas into concrete systems.

I have been fortunate enough to work with several wonderful collaborators over the last several years. I began working with Rob Harrison and Mina Tahmasbi Arashloo on P4All during my first semester at Princeton. They guided me through the world of programmable switches, and they also showed me how to handle all of the uncertainties and struggles that come with research, and how to approach research with confidence. Our first deadline with P4All remains one of the highlights of my Princeton experience. I am also grateful to Devon Loehr and John Sonchack for their invaluable work and encouragement on the Parasol project. Parasol would not have been possible without their expertise in programming languages and willingness to patiently share it with me. I also thank my internship mentors, Sharad Agarwal, Ryan Beckett, and Rachee Singh, for their advice and guidance.

To the members of my Cabernet family, both past and present, I am endlessly grateful for your support and friendship. I am particularly thankful to Yufei Zheng, our shared passion for Formula 1, trips to the driving range, and our many conver-

sations over just about everything are some of my fondest memories of my time at Princeton; to Sophia Yoo, for your much-needed enthusiasm, optimism, and thoughtfulness; to Sata Sengupta, for graciously letting me interrupt your work to have long conversations about life; to Robert MacDavid, for bringing humor and levity to the office, and for making me feel so welcome from the moment I joined the group.

To my friends, who have always been a soft place to land, you have given me so much joy and have been a welcome respite during the stressful times. To Natalie, Angelina, Anja, Wynne, and Anne, thank you for all the the hiking trips, game nights, and Friendsgivings. During the pandemic, our long walks, poker games, tattoos, movie nights, hair escapades, and housewives marathons were the source of endless laughter and comfort. To Sarah and Sam, I will be forever lucky that we found each other all those years ago. You're my home away from home, even when we're thousands of miles away.

Most importantly, my family has been a continuous source of support and love, without which none of this would be possible. To Grandma, Papa, Diane, Debbie, Donna, Dave, Dad, Daniel, Marissa, and Oliver, thank you for being there for me. To Kathryn, Ken, Grace, and Silas, you have kept me grounded throughout this journey, and you are my lifelong best friends. Finally, to Mom, I am forever indebted to you for all of the sacrifices you have made for me. You made me the person I am today, and have loved me unconditionally through it all. Thank you for everything.

The research in this dissertation was supported by the National Science Foundation awards FMiTF-1837030 and CNS-1703493, the Israel Science Foundation grant No. 980/21, and DARPA contracts HR0011-17-C-0047 and HR0011-20-C-0160.

To my family.

Contents

List of Tables

List of Figures

Bibliographic Notes

The material presented in chapter 3 is joint work with Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison, and has been previously published and presented at ACM HotNets 2020 [34] and USENIX NSDI 2022 [33]. The material presented in chapter 4 is joint work with Devon Loehr, John Sonchack, Shir Landau Feibish, Jennifer Rexford, and David Walker, and has appeared in an arXiv paper [35].

Chapter 1

Introduction

Computer networks are the foundation of today's society. Most of our daily lives involve interacting with services that rely on networks (video streaming, big data analysis, cloud gaming, etc.), and we expect the network to securely and efficiently carry out these services. These seemingly simple services that happen almost instantaneously are made possible by a complex web of devices that process user traffic, and the performance of those services is completely dependent on underlying network devices reliably processing traffic at very high speeds.

Early networks only needed to support communication between a small number of hosts. However, in the decades following, the number of applications, beyond just simple communication, reliant on networks exploded—networks were suddenly responsible for applications that require transmitting massive amounts of data, and near instantaneous feedback. In the following sections, we explore several categories of modern applications, explain why traditional networks, designed for basic communication, are not up to the task of supporting these new applications, and the challenges that come with trying to modernize these traditional networks.

1.1 Modern Network Applications

As mentioned above, so much of our society relies on applications supported by networks. Three categories of popular modern applications are:

- **Network management.** Applications utilizing networks are only possible if the networks themselves are functioning properly. Network management applications ensure that the network is running smoothly, and they help network operators identify and locate problems within a network. These applications include telemetry to measure and collect statistics in a network, and automatic adaptation in the network when possible (*e.g.,* active queue management, rerouting around congestion and/or failures, etc.).

- **Services for distributed applications.** As more and more users connected to applications running inside data centers, distributed systems allowed those applications to scale to many users, all accessing those applications simultaneously. The distributed architecture, however, created new challenges that necessitated new functionality in the network, for coordination and improved performance (*e.g.,* caching, chain replication, Paxos, etc.).

- **Middleboxes.** Middleboxes are specialized devices inside the network that add additional functionality, primarily for security and performance benefits. Examples of services they provide are firewalls to block malicious traffic, load balancing to mitigate congestion, network address translation (NAT), and VPN termination.

These applications must scale to millions of users, who expect minimal delay and nearly real-time responses. Running them inside the network, instead of at end hosts (*e.g.,* inside servers in a data center), can make those expectations a reality, by improving response times, throughput and scalability, along with power consumption [73]. Even just offloading parts of these applications to in-network devices can

yield significant benefits, such as, for example, accelerating database queries [40, 72] and neural network inference [60].

Modern applications are very complex, and are constantly changing and developing to meet the needs of users. As such, the network needs to also change to support those developments, and network operators need to have the flexibility and control to facilitate that change. In other words, networks need to be *programmable*. Unfortunately, conventional network devices are not easily customizable, and are not able to keep pace with application innovation.

1.2 Shortcomings of Traditional Networks

Networks are made up of a web of interconnected switches, which are specialized devices meant to forward traffic to the appropriate destinations (*e.g.,* servers, mobile phones, laptops, etc.). Switches were made to be fixed-function—*i.e.,* operators have little control over the functionality of the switch beyond packet forwarding. These switches worked well in early networks, because switches only needed to forward packets; compute-intensive and advanced applications were typically implemented at the end-host, leaving the most simple tasks for inside the network [59]. However, with the advent of applications discussed in 1.1, and the scale and speed at which they must perform, in-network computing is becoming more essential. In order to change fixed-functions switches, though, operators cannot just write a program to change how packets are processed. Changes typically have to be made in the hardware by the switch vendor, which is a time-consuming, lengthy process that makes it nearly impossible to deploy new applications in a timely manner.

Software switches, implemented on CPUs, provide more flexibility and control than fixed-function switches. Packet-processing programs can be written with general-purpose languages, and software switches have access to plenty of resources

and memory. However, processing packets in software is simply too slow; CPUs cannot keep up with modern link speeds (100+Gbps).

Software-defined networking (SDN) was an initial attempt to bring more programmability into the network. SDN separates the data plane (simple packet-forwarding switches) from the control plane (a centralized software program managing the behavior of the network). The goal is for operators to be able to program high-level policies in the control plane, which then get deployed in all of the data-plane devices.

Alongside the introduction of the SDN paradigm was the development of OpenFlow [51]. OpenFlow is a protocol to program flow tables in switches. When packets enter a switch, they can match a particular entry in the table (according to some predefined fields, as specified by the control plane), and operators can program specific actions to take upon a match. OpenFlow was designed to exploit the design features common among switches from various hardware vendors, so that a network of diverse devices can be programmed through a single control plane. This, however, limits the programmability of the switches to only a small set of features. While the control plane is easily programmable through software, the data plane has lagged behind.

1.3 Programmable Network Devices

Programmable devices [20, 25, 39, 52, 55] were developed to address the limitations of traditional, fixed-function and software switches. The advent of programmable network devices has provided network operators the ability to customize the behavior of switches, paving the way for sophisticated applications that run inside the network itself, as described in 1.1. They allow operators to deploy new applications directly on the actual hardware, instead of relying on general-purpose computers (which are too slow) or depending on switch vendors to add desired features to the hardware.

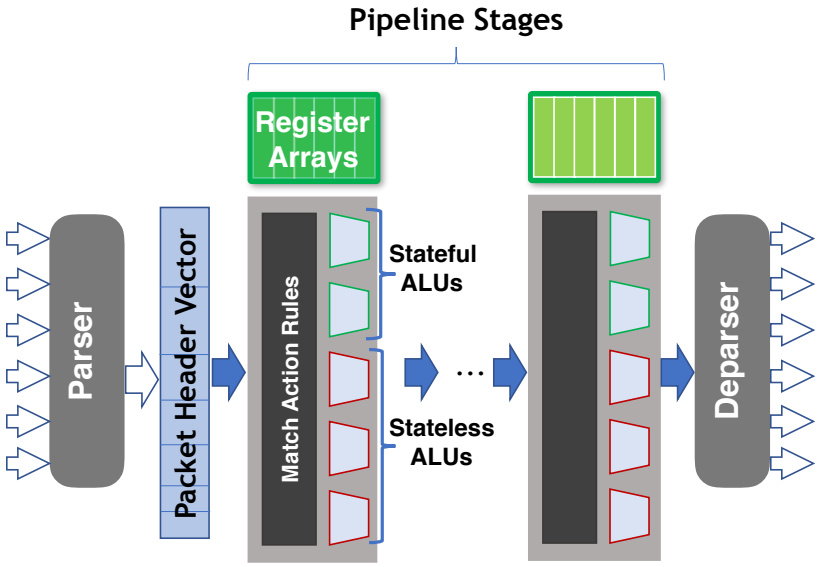


Figure 1.1: Protocol Independent Switch Architecture (PISA).

Most importantly, they provide programmability while also guaranteeing the speeds necessary for modern networks. The switches utilize programmable ASICs, with a specialized architecture designed to process user traffic at high speed.

1.3.1 Programmable Switch Architecture

Programmable switches, like the Intel Tofino [39], typically implement a *Protocol Independent Switch Architecture* (PISA) (Figure 1.1). Such an architecture contains a programmable packet parser, processing pipeline, and deparser. When a packet enters the switch, the parser extracts information from the packet and populates the *Packet Header Vector* (PHV). The PHV contains information from the packet's various fields, such as the source IP, TCP port, *etc.* that are relevant to the switch's task, whether it be routing, monitoring, or load balancing. The PHV also stores additional per-packet data, or *metadata*. Metadata often holds temporary values or intermediate results required by applications deployed on the switch. Finally, the deparser reverses the function of the parser, using the PHV to reconstitute a packet and send it on its way.

Between parser and deparser sits a feed-forward packet-processing pipeline. This pipeline consists of a series of stages, and when a packet travels through the pipeline, it is processed at each stage sequentially. A program may recirculate a packet by sending it back to the beginning (to allow for additional processing), but too much recirculation decreases throughput. Each stage contains a fixed set of resources, which bounds the amount of computation it may perform before the packet moves on to the next stage. The following is a summary of the resources in a PISA switch:

- **Pipeline stages.** The processing pipeline is composed of a fixed number (S) of stages. All actions executed in a stage happen in parallel, so dependent actions must be in separate stages.
- **Packet header vector (PHV).** The PHV that carries information from packet fields and additional per-packet metadata through the pipeline has a fixed width (P bits).
- **Registers.** A stage is associated with M bits of registers that serve as persistent memory.
- **Match-action rules.** Each stage stores match-action rules in either TCAM or SRAM (T bits). When a packet's fields (stored in the PHV) match a rule, an associated action is performed.
- **ALUs.** Actions are performed by ALUs associated with a stage. Each stage has F stateful ALUs (that perform actions requiring registers) and L stateless ALUs (that do not).
- **Hash units.** Each stage can perform N hashes at once.
- **Recirculation bandwidth.** A program may recirculate a packet by sending it back to the beginning of the pipeline once it leaves the last stage. If too many packets are recirculated, throughput is decreased.

As an example of how an application maps to each of these components, assume a program that counts the number of packets seen for distinct source IP addresses. When a packet arrives at the switch, it will be parsed according to programmer-defined rules, and part of the PHV will be used to store the source IP address from the packet header. Match-action rules decide what action to perform on each packet—if the switch has seen the IP address before, the packet will match a rule to increment its corresponding counter. The registers are used to store the counts, as well as the associated IP address. The ALUs are programmed to perform the appropriate actions on registers—either retrieving a stored count or updating a counter. The programmer can use hash units to calculate the index of the registers to access for a particular packet. Lastly, the packet could be recirculated back to the beginning of the pipeline for further processing, at the cost of reduced throughput.

1.3.2 Resource-Constrained Hardware

The power of programmable switches comes at a price: the hardware architecture is notoriously restrictive. This ensures that applications running on the switch (*e.g.,* load balancing, telemetry, etc.) can keep up with today's link speeds and that processing at the switch does not become a bottleneck, but it also means that programmers often run up against the hardware limits of the switch. Even when switches only implement simple functionality, resources, such as registers for tallying traffic statistics, are at a premium. For example, in the simple counting example described in 1.3.1, the switch does not have enough registers to store counts for every possible IP address it may see.

As a result, programmers have employed several strategies to design applications that can work within the restrictive hardware constraints, each of which has associated advantages and disadvantages. Below are three commonly used strategies, which can be employed by themselves or in combination with each other.

- **Sampling.** Instead of measuring or recording every packet processed by a switch, a program can instead sample a small subset of packets. In the IP counting example in 1.3.1, the switch would only track a small percentage of the total IP addresses it sees. While a sample of packets can be used to gather statistics about user traffic, if the sampling rate is too low, the resulting measurements will likely not be accurately representative.

- **Approximation.** Approximate, space-saving data structures are often used to alleviate pressure on switch resources because they keep *approximate* statistics, using just a fraction of the memory required for exact measurements. These structures can provide highly accurate estimations, but they are sensitive to the resources allocated to them—the less memory they use, the less accurate the approximations will be.

- **Switch + General-Purpose Compute.** Instead of only relying on the switch, applications can utilize general-purpose compute in conjunction with programmable switches. As mentioned above, general-purpose compute is significantly slower than a programmable switch. However, programs running a switch can send a small number of packets to be processed in software without significant service disruption. The switch, for example, can function as a cache—the majority of traffic is processed in the data plane, and the rest is sent for further processing in software. If the amount of traffic sent to software is too large, though, the application can become prohibitively slow, which negates the benefit of running an application in the switch.

Writing applications for resource-constrained hardware requires a programmer to make a plethora of decisions. Data-plane programs can have various parameters that may significantly affect the program accuracy and performance. For example, a programmer may use sampling to measure the counts for a subset of the IP addresses

seen by the switch. The programmer must decide on a sampling rate—too large, and the switch resources will not be able to support it, but a too small rate will not yield useful information. Similarly with an approximate data structure, the programmer must allocate enough memory to get accurate enough estimations, but small enough to fit within the resource limitations. Lastly, if the programmer takes advantage of general-purpose compute, the switch must store enough of the IP counts, so as not to overwhelm the CPU and reduce throughput, without exhausting switch resources.

Determining parameter values is by itself a difficult problem, but it is compounded when multiple applications are sharing the same switch. For example, it may be useful have a switch that forwards traffic, along with collecting measurements and running a firewall, which drops malicious traffic. Writing these applications gets exponentially more challenging when there are multiple structures, because allocated resources can be a zero-sum game: giving resources to one structure leaves less for others. The programmer then has to reason about how to divide resources among them and how that will affect the performance of each structure individually, and the application as a whole.

1.4 P4

Because of the unique architecture, general-purpose programming languages like C, Java, or Python are unsuitable for programmable switches. Instead, switches require a domain-specific language that better fits the capabilities of the hardware. To this end, researchers have developed a language for these switches: P4 [10, 57].

P4 has quickly become a key language for programming network data planes. Using P4, programmers can define their own packet headers and specify how the data plane should parse and process them [10]. The P4 language helps manage data-plane resources by providing a layer of abstraction above PISA. A P4 compiler

maps these higher-level abstractions down to the PISA architecture and organizes the computation into stages.

The P4 language has a number of constructs that correspond to the hardware architecture. The following is a summary of the most commonly used constructs:

- **Control flow.** In a P4 program, the *control block* describes the processing to be done on each packet. It contains a sequences of actions, and these actions must fit within the number of pipeline stages on the target switch.

- **Metadata.** Programmers can define *metadata* fields to store data pertaining to a packet. Metadata fields can serve as variables to store the result of intermediate computations, and are part of the PHV.

- **Register arrays + stateful actions.** If a program requires persistent memory, programmers use *register arrays* to define a data structure (*e.g.,* a hash table), made up of registers in a stage. Additionally, stateful actions are used to define what actions can be taken on a register (*e.g.,* incrementing or decrementing the stored value, comparing the stored value to a metadata value, etc.).

- **Tables.** Programmers can use *tables* to outline what fields in a packet header to match on, and the corresponding actions to perform upon a match. These tables are implemented in TCAM on the switch hardware.

- **Actions.** Actions describe how packet header fields and metadata are manipulated. These can include operations such as addition, subtraction, and bit shifts, and are performed by ALUs.

- **Hash externs.** P4 provides support for externs, which can be any hardware-specific construct. One of the most widely used externs are *hash functions*, which are performed with a switch's hash units.

Using these constructs, programmers can customize the behavior of a programmable switch and implement sophisticated applications. In practice, however, the architecture of programmable switches introduces challenges that P4 cannot sufficiently address.

1.4.1 Programming Challenges

As discussed in section 1.3.1, programmable data planes have a restrictive architecture with limited resources, to guarantee line-rate processing of traffic. The restrictiveness of the hardware is also very apparent in the language most commonly used to program switches (P4)—it is very low-level, which forces programmers to specify the hardware components required by an application. This unfortunately puts the burden of reasoning about resource allocation on programmers. They have to understand how each component of an application will get mapped to specific hardware resources and decide how resources should be split among the various components for the best performance, given a particular environment or workload.

Programs written for programmable switches have very strict requirements—they *must* fit within the resources of the hardware, and they *must* run at line rate. To this end, programmable switch compilers are responsible for guaranteeing that a program meets these conditions. If they do not, the program will simply fail to compile, often giving little to no guidance as to where the problem may be. This is in stark contrast to other types of programming (*e.g.,* programming for general-purpose compute), where programmers can write applications that use any number of resources, which don't necessarily need to be known at compile time.

This programming paradigm results in a difficult and tedious process. While writing an application, programmers must hard code the resource allocation and parameter values (described in 1.3.2) into their program. They then must check if the allocation is valid given the hardware layout by compiling the program (which

Data Structure	Used in
Key-value store/ hash table	Precision [5], Sonata [29], Network-Wide HH [31], Carpe [32], Sketchvisor [37], LinearRoad [40], NetChain [41], NetCache [42], FlowRadar [48], HashPipe [66], Elastic Sketch [83]
Hash-based matrix (Sketch)	AROMA [4], Sketchvisor [37], Sketchlearn [38], NetCache [42], Nitrosketch [49], UnivMon [50], Sharma et al. [63], Fair Queueing [64], Elastic Sketch [83]
Bloom filter [9]	NetCache [42], FlowRadar [48], SilkRoad [53], Sharma et al. [63]
Multi-value table	BeauCoup [17], Blink [36]
Sliding window sketch	PINT [6], ConQuest [18]
Ring buffer	NetLock [84], Netseer [87]

Table 1.1: PISA data structures.

may take hours or even days [28]). Unfortunately, the first version of an application is rarely the best one. Consequently, applications often require many revisions and sometimes weeks' worth of tweaking parameter values and testing to obtain a variant of the initial program that compiles, fits within hardware constraints, and performs well.

As a result, code is not often reuseable: a hash table to measure counts of IP addresses, that fits just fine on a switch alongside a table for IP forwarding, is suddenly too large when a firewall is added. To squeeze the hash table in, programmers may have to rewrite the internals of the table, manually adjusting the number or sizes of the hash entries. Beyond the overhead of recompiling programs to see if they fit on the hardware, there is also a significant overhead in just rewriting the program for a specific context. P4 is extremely restrictive, as it does not allow any loops, so to increase the size of a hash table or add an additional table, the programmer would have to manually add all of the code, instead of just incrementing the number of loop iterations.

The lack of reusability is a widespread problem; it's very common for the same data structures to be used in a variety of applications. As an example, Table 1.1 lists

six popular data structures and a selection of applications in which they appear. Instead of being able to reuse existing code for a hash table, for example, programmers will likely have to manually adjust that same data structure for each individual application. P4 as a language cannot support the flexibility required for programmable data planes, and as a result, it shifts the burden of making low-level decisions onto programmers. These difficulties are not specific to P4; they appear in other languages made for the data plane ([27, 65, 68]).

1.5 Decoupling Language and Hardware

Conventional methods of programming switches do not sufficiently meet the needs of programmers, and they often significantly lengthen the time to develop and deploy new applications to programmable switches. The core problem with programmable switches is that the programming language is too tightly coupled to the switch hardware. While domain-specific languages are necessary for such specialized hardware, they need to have a high enough level of abstraction so that the language is actually practical, and doesn't require weeks or months of rewriting programs just to get them to compile to the hardware.

To more easily and effectively program switches, we need an abstraction that separates the code from the hardware. This will make programs more modular and reusable, as well as break the cyclical development process, in which programmers continuously write, test, and rewrite their code until it compiles and performs well.

There are two main features necessary to achieve this separation:

1. **Elasticity.** Programmers should not have to explicitly define in their code how their program is laid out in the hardware, or how to divide limited resources amongst program components. Instead, the programming language should for allow for *parameterized* programs, where any features of a program can be ex-

pressed as parameters. In other words, programs should be *elastic*—parameter values can stretch or shrink according to the particular application, without the programmer needing to rewrite anything.

2. **Automated optimization.** Although the programmer should not have to reason about which parameter values are feasible with the hardware or about resource allocation among program components, those things still need to be known at compile time. If that responsibility no longer lies with the programmer, there needs to be a system that automates what programmers are currently doing by hand. This system should automatically set program parameters, and those parameter values should be optimized to give the best accuracy and performance, as defined by the programmer.

1.6 Contributions

This dissertation focuses on developing practical abstractions to simplify the process of programming data-plane applications. We design these abstractions by (1) adding new language constructs that allow programs to be generalized to various environments, and (2) building optimization systems that adjust programs for specific contexts, lifting that burden from the programmer. The new language constructs allow applications to be parameterized, so program features typically rewritten manually for different environments (*e.g.,* resource allocation) can be easily adjusted by setting a parameter value. The optimization system further simplifies the process by automatically finding the best parameter values for programmer-defined performance goals and particular traffic distributions. We demonstrate that our abstractions can express a wide range of applications, and our optimization produces programs that are at least as performant as hand-optimized programs.

1.6.1 Elastic Language Abstraction

As discussed in 1.4.1, data-plane programming typically involves a tedious trial-and-error process because of the restrictiveness of both the language and the hardware. This manual process of tweaking the *internal* details of data structures, and checking whether the resulting structures satisfy *global* constraints, is inherently non-modular: programmers tasked with implementing separate applications cannot do so independently. Indeed, while the same data structures appear again and again (see Table 1.1 for a selection), their parameters must be tuned to satisfy varying resource constraints and accuracy goals, making it difficult to reuse these structures for different hardware targets or applications.

We extend data-plane programming languages with the ability to write *elastic* programs. An elastic program is one whose various algorithmic decisions (*e.g.,* sampling rate, data structure size, etc.) can be expressed as parameters that can change according to desired accuracy and available resources. Elastic structures can "shrink" to accommodate other structures sharing the same resources, or they can "stretch" arbitrarily to fill available space.

Using the IP address counting application as an example, if this is the only program running on the switch, it can utilize all of the available memory on the switch. If the programmer decides to add a second application, perhaps a stateful firewall, the elastic IP address counting structure can shrink, to free up memory for the firewall. The structures do not need to be rewritten for each context; only the parameter values need to change.

To realize this concept of elasticity, we implement backward-compatible extensions to P4 in the P4All system, and to Lucid [68] in the Parasol system. Lucid is a high-level, event-based data-plane programming language. In particular, we utilize the following features:

- **Symbolic values.** Symbolic values function as placeholders that may take on any value of the given type. Once declared, a symbolic is used in the same way as a compile-time constant. For the IP address counting example, symbolic values could include the size of the structure used to store counts for each address, and a timeout threshold to evict expired entries.

- **Bounded loops.** While programmable switch hardware cannot support loops if the number of iterations are not known at compile time, they can support loops bounded by symbolic values, which work as compile-time constants. Because parameter values are not necessarily known by the programmer while they are writing an application, loops bounded by symbolics allow programmers to define actions that should be repeated over elastic structures.

- **Objective functions.** There are many different possible values for each parameter in a particular program, and the values can directly affect the performance and accuracy of a program. While we design a system to automatically set those parameters (1.6.2), we still need some way for the programmer to define performance or accuracy for their specific program. Objective functions provide a principled way for the programmer to control the setting of parameter values. If the programmer is counting the number of times particular IP addresses are seen, their objective function may be the accuracy of the counts stored in the switch, with the goal of maximizing the achievable accuracy.

We evaluate our language components by developing a number of reusable, elastic structures and building several elastic applications using these structures. We show that both P4All and Parasol are expressive enough to implement programs with a wide variety of parameter types (*e.g.,* memory layout, data structure selection, threshold values, etc.) and objectives (*e.g.,* accuracy, recirculation overhead, collision rate, etc.).

1.6.2 Automated Optimization of Parameters

When writing elastic programs, programmers use symbolic values as placeholders. While programmers do not have to decide what concrete values those symbolics will eventually take, those values still need to be known at compile time, because of the restrictive hardware architecture—if a program does not adhere to the constraints, it cannot be compiled to the hardware. As such, we need a system to automatically set parameter values, according to a programmer-defined objective.

We implement two distinct systems for optimizing parameters. In P4All, we generate and solve an integer-linear program (ILP), whose constraints represent the resource constraints on the switch, and the variables correspond to the symbolic values in the application code. Objective functions in P4All are closed-form equations that express application performance or accuracy as a function of the symbolic values, and the optimal parameter values are those that either minimize or maximize the objective function.

Closed-form objective functions provide provable guarantees on worst-case performance, but they are often difficult to derive in practice. The relationship of symbolic values to performance is not always clear, particularly in complex applications that may have multiple data structures interacting with each other. For these cases, we need an alternative method of describing objective functions and optimizing application performance.

In contrast to P4All, we implement a system in Parasol that uses an iterative search algorithm to automatically optimize parameters. In each iteration, a search algorithm selects a concrete value for each symbolic value, and the resulting program is then simulated on a provided traffic trace, which allows a more tailored optimizations than would be possible from relying solely on static, workload-independent quantities such as switch resource usage. Parasol objectives come in two parts. The first part measures arbitrary aspects of a program executing in simulation mode over

an example traffic trace. The second part computes an arbitrary, user-defined score based on those measurements. Both parts are implemented in Python rather than the more limited languages of switch data planes. At the end of the optimization process, the Parasol system returns parameter values that yield the best score, as measured during simulations.

We evaluate both strategies by developing a number of data-plane programs with various parameters and objective functions. We compile the resulting programs to a real target (the Intel Tofino) to verify that the optimizers produce valid program code. Our experiments find that all applications could be optimized in under two hours, and the solutions produced by the optimizers not only complied with the resource constraints of the hardware, but were comparable in performance to hand-optimized P4 code.

Chapter 2

Writing Data-Plane Applications

In this chapter, we provide examples of data structures commonly used for programmable switch applications. We also introduce the constructs of P4, and show an example implementation of an approximate data structure (a count-min sketch) to illustrate the challenges of data-plane programming.

2.1 Approximate Data Structures

Approximate, space-saving data structures are commonly used in resource-constrained environments, such as data-plane programming, because they are designed to keep accurate approximate statistics using a small amount of memory. In this section, we explore two structures: the count-min sketch [22] and Precision [5].

A count-min sketch (CMS [22]) is a probabilistic data structure that uses multiple hash functions to keep approximate frequencies for a stream of items. Intuitively, the CMS is a two-dimensional array of w columns and r rows. For each packet (x) that enters the switch, its flow ID (f_x) is hashed using r different hash functions ($\{h_i\}$), one for each row. The output of the hash function determines which column in a row is incremented for f_x, so the output of the functions ranges from $(1 \dots w)$. For example, in the second row of the CMS, hash function h_2 determines that column

($h_2(f_x)$) is incremented. To approximate the number of times flow f_x has been seen, one computes the minimum of the values stored in columns $h_i(f_x)$ for all r rows.

The CMS can only keep approximate counts, instead of exact, because of hash collisions. Suppose the sketch has ten columns, but it is tracking the frequencies of a hundred flows. Multiple flows will get hashed to the same column, causing an overestimation. We can reduce the number of collisions, and consequently increase the accuracy of the estimations, by increasing the number of columns.

We can also improve the confidence of the estimations by adding more rows to the sketch. Flows that collide in one row are unlikely to collide in every other row, because each row uses a different hash function.

Precision [5] is another type of data structure designed for efficient heavy hitter detection (*i.e.,* identification of large flows). Similar to the CMS, Precision is two dimensional array with r rows and w columns. There are r independent hash functions that are used to compute an index for each row. Unlike the CMS, Precision stores both a key (*e.g.,* flow ID) and a counter for each key, and the count for a particular flow ID is only stored in a single row.

When a packet enters the switch, we first check if the flow is already stored in the Precision structure, by computing the hash functions and checking the keys at the corresponding indexes. If the flow is already in the structure, we can increment it count. If the flow is not in the structure, we have to decide if it should be inserted, and if so, where it should be added. The Precision algorithm chooses to *probabilistically* evict the entry with the smallest count and insert a new flow in its place. Items with larger counts are less likely to get evicted, allowing Precision to accurately identify heavy hitters.

The probabilistic insertion serves two purposes:

1. In a network, there are typically many small flows, and a smaller number of large flows. If new flows are always inserted, large flows are likely to get evicted

often by small flows, making it difficult to maintain accurate counts. Heavy hitters are more likely to stay in the structure if we only probabilistically add a fraction of new flows.

2. Adding a new entry requires a recirculation—the packet must go back to the beginning of the processing pipeline and traverse it again. If too many packets are recirculated, throughput begins to degrade. Probabilistic insertion ensures that throughput is not significantly affected.

The performance of Precision, like the CMS, improves as the structure grows larger. As more memory is allocated the structure, it can track frequencies of more flows. This reduces the number of collisions and evictions, and improves the accuracy of the counters.

The performance of these data structures is dependent on their resource usage—the more memory allocated to them, the more accurate their statistics. Their implementation, then, is seemingly quite simple. Programmers should allocate all of the switch resources to whichever structure they use. Unfortunately, most switch applications are much more complex than just implementing a single data structure. Switches are responsible for a variety of tasks, including both forwarding and collecting statistics. One application cannot take up all of the hardware resources, because it then leaves no room for other applications. Even if the switch only needed to perform a single application, many of them require multiple data structures. In the next section, we detail an example of an application that is implemented with two distinct structures.

2.2 Multi-Structure Applications

We provide a motivating example application that one might wish to deploy on a programmable data plane: a load-balancing cache designed to accelerate response

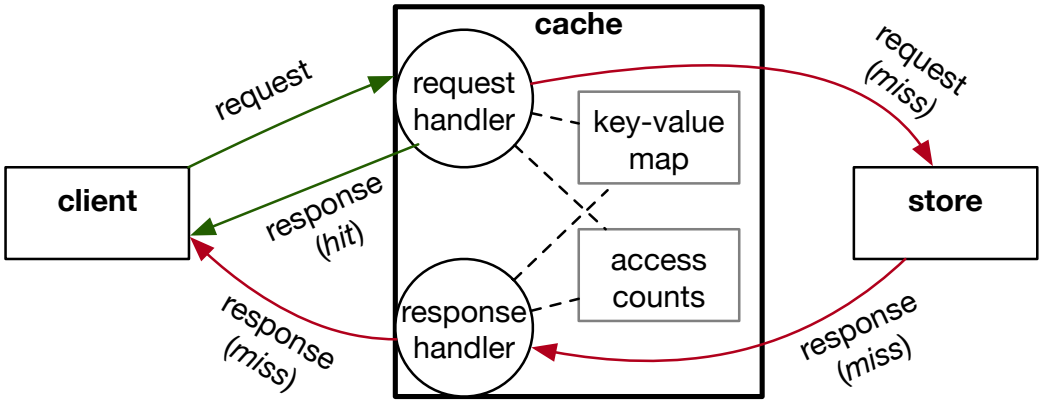


Figure 2.1: Data-plane cache.

times for web services, inspired by NetCache [42]. The structure of the cache is illustrated in Figure 2.1. The cache reduces load on storage servers by directly serving requests for the most popular keys, and forwarding only cache misses to the servers.

The cache operates by storing key/value pairs in a hash table on a switch. When a request arrives, the switch first checks to see if the key is in the table; if it is, the switch simply retrieves the value and sends it back to the requester. Otherwise, the switch forwards the request to the appropriate storage server. When the response arrives, the switch forwards it to the client and optionally caches the entry.

To maximize efficiency, and to best utilize the limited available memory, the cache should serve requests for the most popular keys. Because popularity may change over time, the switch dynamically updates its cache to remove less popular keys in favor of more popular ones. To enable this, the switch tracks statistics about the popularity of keys *not* stored in the cache using a second data structure: a compact, approximate counter (*e.g.,* a count-min sketch (CMS)).

The resource allocation for the data-plane cache is not straightforward. Both the hash table used for the key-value store and the CMS perform better with more resources—the hash table can cache more keys, and the CMS has more accurate counts. However, allocating resources is a zero-sum game. Because they are so

limited, giving resources to one structure means taking it away from another. There is an inherent tension, then, of needing to give structures just enough resources to perform well, but not so much that other structures suffer.

The performance impact of resource allocation is even less clear—is it better to store more keys in the key-value store, or more accurately identify popular keys in the CMS? Allocating the resources of a programmable switch is like solving a jigsaw puzzle, trying to get every structure to fit onto the switch in a configuration that yields the best performance. This puzzle can become even more complex than resource allocation. In the data-plane cache, we must decide if a key-value store and CMS are even the right data structures. One could imagine implementing a cache with Precision instead, storing keys and values, along with their counts.

All of these decisions must be made by the programmer, by hand, because data-plane programming languages are so low-level. In the next section, we walk through an example implementation of a CMS in P4, and show that the language creates these challenges for programmers by requiring them to encode resource allocation into their application code.

2.3 Programming Approximate Structures

To illustrate the constructs of P4 and the shortcomings of the language, we show a fragment of a P4 program that implements a CMS (Figure 2.2, included at the end of this chapter). The primary actions for a CMS are (1) executing the hash functions, (2) incrementing the counts stored in the sketch, and (3) computing the minimum value across sketch rows. Each of these actions require resources: metadata for storing the result of hash functions and the counts in the sketch (to later compute the minimum), register arrays for storing information, and ALUs to execute the actions. Each resource used must be explicitly declared in the code.

We first declare the metadata used by the CMS to store a count at a particular index (*i.e.,* a hash of a flow ID).

```
struct custom_metadata_t {
  bit<32> index0;
  bit<32> count0;
  ...
  bit<32> index3;
  bit<32> count3; }
```

We then declare the low-level data structures (registers) that actually make up the CMS—four rows ($r = 4$) of columns ($w = 2048$) that can each store values represented by 32 bits.

```
register<bit<32>>(2048) counter0;
...
register<bit<32>>(2048) counter3;
```

We also declare the actions for hashing/incrementing at each row and for updating the metadata designed to store the global minimum.

```
action GetAndIncrStoredValue0() { ... }
...
action GetAndIncrStoredValue3() { ... }
action Min_0(){meta.min = meta.count0;}
...
action Min_3(){ . . . }
```

The hashing action is a complex action containing several atomic actions: (1) an action to hash the key to an index into a register array, (2) an action to increment the count found at the index, and (3) an action to write the result to metadata for use later in finding the global minimum. Such multi-part actions can demand a number of resources, including several ALUs and hashing units.

In the `apply` fragment of control block in the P4 program, the program first executes all the hash actions, computing and storing counts for each hash function, and then compares those counts to each other, looking for the minimal one.

```
apply {
  meta.min = 0xffffffff; /*initialize global min*/
  /* Compute hash indexes and retrieve stored counters */
  GetAndIncrStoredValue0();
  ...
  GetAndIncrStoredValue3();
  /* Compute minimum */
  if (meta.count0 < meta.min) { Min_0();}
  ...
  if (meta.count3 < meta.min) { Min_3();}
} }
```

Upon reviewing this code, some of the deficiencies of P4 are immediately apparent. First, there is a great deal of repeated code: Repeated data-structure definitions, action definitions, and invocations of those action definitions in the apply segment of the program. Good programming languages make it possible to avoid repeated code by allowing programmers to craft reusable abstractions. Avoiding repetition in programming has many good properties, including the fact that when errors occur or when changes need to be made, they only need to be fixed/made in one place. This not only saves time but helps avoid subsequent errors. Effective abstractions also help programmers change the number or nature of the repetitions easily. Unfortunately, P4 is missing such abstractions.

One might also notice that the programmer had to choose an appropriate size for the data structure (2048 columns and 4 rows). Because picking a resource allocation that adheres to resource constraints is so difficult, the process of determining data structure sizes is a tedious one, involving a lengthy cycle of guessing a resource allocation, and compiling and rewriting the code.

These decisions and challenges exist because of the nature of data-plane programming. In order to be able to process large amounts of traffic and guarantee that processing executes at line-rate, the switch architecture, unlike general-purpose hardware, is designed in such a way that requires programmers to reason about resource utilization and other low-level details. In the next chapter, we detail a system that raises the level of abstraction for data-plane programming by separating resource allocation from program code.

```
1  struct custom_metadata_t {
2    bit<32> cachedKey;
3    bit<32> cachedTime;
4    bit<32> cachedValue;
5    bit<32> min;
6    bit<32> index0;
7    bit<32> count0;
8    ...
9    bit<32> index3;
10   bit<32> count3; }
11 control Ingress( ... ) {
12   /* A register array for cache hash table */
13   register<bit<32>>(1024) keyValue;
14   /* A register array for each CMS row */
15   register<bit<32>>(2048) counter0;
16   ...
17   register<bit<32>>(2048) counter3;
18
19   /* An action to check if key in cache and if
         entry expired */
20   action CheckCachedKey() { ... }
21
22   /* Actions to update each CMS row */
23   action GetAndIncrStoredValue0() { ... }
24   ...
25   action GetAndIncrStoredValue3() { ... }
26   /* An action to set the minimum */
27   action Min_0(){meta.min = meta.count0;}
28   ...
29   action Min_3(){ . . . }
30   /* Execute the following on each packet */
31   apply {
32     /* Check if key in cache, and add to cache
          if expired */
33     CheckCachedKey();
34
35     meta.min = 0xffffffff; /*initialize global
          min*/
36     /* Compute hash indexes and retrieve
          stored counters */
37     GetAndIncrStoredValue0();
38     ...
39     GetAndIncrStoredValue3();
40     /* Compute minimum */
41     if (meta.count0 < meta.min) { Min_0();}
42     ...
43     if (meta.count3 < meta.min) { Min_3();}
44     } }
```

Figure 2.2: Cache in P4.

Chapter 3

P4All: Modular Switch Programming Under Resource Constraints

This chapter focuses on the extension of P4 with the ability to write *elastic* programs. An elastic program is a single, compact program that can "stretch" to make use of available hardware resources or "contract" to squeeze in beside other applications. Elastic programs can be constructed from any number of elastic components that each stretch arbitrarily to fill available space. An elastic data-plane cache program, for example, may be constructed from an elastic count-min sketch and an elastic key-value store. The programmer can control the relative stretch of these modules by specifying an objective function that the optimizer should maximize. For example, the caching application could maximize the cache "hit rate" by prioritizing memory allocation for the key-value store (to store more of the "hot" keys) while ensuring that enough remains for the CMS to produce sufficiently accurate estimates of key popularity. In addition to memory, programs could simultaneously maximize the use of other switch resources such as available processing units and pipeline stages.

To implement these elastic programs, we present P4All, a backward-compatible extension of the P4 language with several additional features: (1) symbolic values, (2) symbolic arrays, (3) bounded loops with iteration counts governed by symbolic values, and (4) local objective functions for data structures. Symbolic values make the sizes of arrays and other state flexible, allowing them to stretch as needed. Loops indexed by symbolic values make it possible to construct operations over elastic data structures. Objective functions provide a principled way for the programmer to describe the relative gain/loss from growing/shrinking individual data structures. Global optimization criteria make it possible to weight the relative importance of each structure or application residing on a shared device.

We have implemented an optimizer for P4All that operates in two main phases. First, it computes an upper bound on the number of possible iterations of loops, so it can produce a simpler optimization problem over unrolled, loop-free code. This upper bound is computed by conservatively analyzing the dependency structure of the loop bodies and their resource utilization. Next, the optimizer unrolls the loops to those bounds and generates a constraint system that optimizes the resource utilization of the loop-free code for a particular target. We use the Intel Tofino chip as our target. We evaluate our system by developing a number of reusable, elastic structures and building several elastic applications using these structures. Our experiments show that the P4All optimizer runs in a matter of minutes (or less) and produces P4 programs that are competitive with hand-optimized code.

3.1 Language Extension

A key challenge with data-plane programming that differentiates it from general-purpose languages is the manifestation of hardware constraints in application code. Programmers are responsible for understanding the hardware and how application

components will get mapped to physical resources. In this section, we detail the extensions we make to an existing data-plane programming language (P4) to raise the level of abstraction and decouple the language from the hardware.

P4All improves upon P4 by making it possible to construct and manipulate *elastic data structures*. These data structures may be developed modularly and combined, off-the-shelf, to build efficient new applications. In this section, we illustrate language features by building an elastic data-plane cache, with a CMS to track key popularity and a key-value store to cache popular keys. We include code fragments for a data-plane cache implemented in P4All at the end of this chapter, in Figure 3.7.

3.1.1 Symbolic Values

The first language extension we make is the addition of symbolic values that allow users to write elastic, parameterized programs. Symbolic values control the "stretch" of the structure. In the case of the cache there are three such parameters: (1) the number of rows in the sketch (*i.e.,* the number of hash functions), (2) the number of columns (*i.e.,* the range of the hash), and (3) the number of entries in the key-value store. Such parameters are defined as *symbolic values*:

```
symbolic rows;
symbolic cols;
symbolic cacheEntries;
```

Symbolic integers like `rows`, `cols`, and `cacheEntries` should be thought of as "some integer"—they are placeholders that are determined (and optimized for) at compile time. In other words, as in other general-purpose, solver-aided languages like Boogie [46], Sketch [67], or Rosette [74], the programmer leaves the choice of value up to the P4All optimizer.

Often, programmers know constraints that are unknown to the optimizer. For instance, programmer experience might suggest that count-min sketches with more than

four hash functions offer diminishing returns. We extend P4 with assume statements in P4All, to allow users to explicitly specify bounds on symbolic values.

Such constraints may be written as follows:

```
assume 0 < rows && rows <= 4;
```

An assume statement is related to the familiar assert statement found in languages like C. However, an assert statement *fails* (causing program termination) when its underlying condition evaluates to false. An assume statement, in contrast, always *succeeds*, but adds constraints to the system, guaranteeing the execution can depend upon the conditions assumed.

3.1.2 Bounded Loops and Symbolic Arrays

Programming with P4 is often a tedious process in part because of the repeated code—definitions for data structures and actions on those structures are often repeated multiple times, depending on the size of the structure. For example, each row in a CMS is implemented as a separate register array, and the action to execute a hash function and increment a value stored in the sketch must be declared for every row, despite the actions being almost identical.

To reduce this repetition, we introduce *bounded loops* and *symbolic arrays* as extensions to P4. These loops and arrays are bounded by symbolic values, and are unrolled before compiling to hardware.

As an example, we can define a CMS using symbolic arrays:

```
register<bit<32>>(cols)[rows] cms;
```

In this declaration, we have a symbolic array `cms`, which contains `rows` instances of the register array type. Each register array holds `cols` instances of 32-bit values.

One can also define metadata as symbolic arrays. For instance, for each row of the CMS, we need metadata to record an index and count for that row. To do so,

we define symbolic arrays of metadata, where each element of each array is a 32-bit field. The arrays each contain `rows` items.

```
bit<32>[rows] index;
bit<32>[rows] count;
```

Because elastic data structures can stretch or contract to fit available resources, operations over those data structures must do more or less work in a corresponding fashion. To accommodate such variation, P4All extends P4 with loops whose iteration count may be controlled by symbolic values.

The CMS of our running example consists of two operations. The first operation hashes the input `rows` times, incrementing the result found in the CMS at that location, and storing the result in the metadata. The second iterates over this metadata to compute the overall minimum found at all hash locations. In P4All, each operation is implemented using bounded loops and is encapsulated in its own control block. The code below illustrates these operations.

```
/* Actions used in control segments */
action incr()[int i] { ... }
action min()[int i] { ... }
/* Hash and increment */
control hash_inc( ... ) {
  apply {
    for (i < rows) {
      incr()[i]; } } }
/* Find global minimum */
control find_min( ... ) {
  apply {
    for (i < rows) {
      if (meta.count[i] < meta.min) {
        min()[i]; } } } }
```

Each action used in bounded loop has an additional parameter (`[int i]`) that is the index of the loop. These simple symbolic iterations (`for i < rows`) iterate from zero up to the symbolic bound (`rows`), incrementing the index by one each time. The overarching data-plane cache application can now call each control block.

```
control Cache( ... ) {
  apply {
    hash_inc.apply(...);
    find_min.apply(...);
    ... } }
```

3.2 Objective Functions

Elastic parameters in data-plane programs are valid for a range of values, and although they allow for more generalized programs, their values still need to be known at compile time, because of the nature of switch hardware. However, the programmer should not have to reason about parameter values themselves, so we build a system to automatically set them.

Parameter values affect application performance, so they should be set such that they yield the best possible performance. We cannot infer performance just from the application code, so we rely on user-supplied objective functions that define performance goals of a particular application. In P4All, we provide ways for programmers to write an objective function in the program, as part of the code.

P4All utilizes an integer-linear program (ILP) for optimization, where constraints of ILP are constraints of hardware, and variables correspond to symbolic values that control hardware resource usage. As such, the objectives are closed-form formulas that describe performance in terms of symbolic values, and can be plugged into the ILP solver. The solver finds concrete instances of each symbolic value such that the objective function is either minimized or maximized.

We extend P4 with the ability to write these formulas, and specify whether they should be minimized or maximized. The P4All backend, written in Python, uses Gurobi [30] as an ILP solver, so objective functions are written in Python syntax. In the data-plane cache example, we can use the hit rate (that is, the ratio of cache hits to cache accesses) as the performance goal for the key-value store. Suppose the key-value store has k items. The probability of a request to the i^{th} most popular item is $\frac{1}{i^\alpha}$ [12]. In this case, α is a workload-dependent parameter that captures the amount of skew in the distribution. Then, for k items, the probability of a cache hit is the sum of the probabilities for each item in the key-value store: $\sum_{i=1}^{k} \frac{1}{i^\alpha}$. Hence, in P4All, for $\alpha = 1$, we might define the following objective function:

```
objective kvObj {
  function: sum(map(lambda y: 1.0/y,range(1,cacheEntries+1))) }
```

Similarly, we can define the CMS error, ϵ, in terms of the number of columns, w, in the sketch. We can set $w = 3(1/\epsilon)^{1/\alpha}$ [23], where α is a workload-dependent parameter. The number of rows in the CMS does not affect ϵ, so we may choose to leave it out of the objective function. However, we can incorporate constraints to guarantee a minimum number of rows, because the number of rows is used to determine a bound on the confidence of the estimations. For $\alpha = 1$, this objective function is `3.0/cols`.

```
objective cmsObj {
    function: 3.0/cols; }
```

In the data-plane cache, both the hit rate of the key-value store and the error rate of the CMS will affect the overall application performance, and the programmer must decide if either data structure should receive a higher proportion of the resources. If the CMS is prioritized, it can more accurately identify heavy hitters. However, the key-value store may not have sufficient space to store the frequently requested items.

Module	Symbolic values	Intuition	Objective Function
Key-value store/ hash table	Number of rows k	NetCache [42]: Maximize cache hits	maximize $\sum_{i=1}^{k} \frac{1}{i^{\alpha}}$
Hash-based matrix (Sketch)	Num rows d, num columns w	NetCache [42] (CMS): Minimize heavy hitter detection error	minimize $\epsilon = (\frac{3}{w})^{\alpha}$
Bloom filter [9]	Num bits m, num hash functions k	NetCache [42]: Minimize false positives. Expected number of items in stream n	minimize $(1 - e^{-\frac{kn}{m}})^k$
Multi-value table	Number of rows k	BeauCoup [17]: Minimize collisions. BeauCoup parameter set B; Probability to insert to table p $= f(\alpha, B)$; Expected number of items in stream n	minimize $(\frac{1}{k})^{n \cdot p}$
Sliding window sketch	Num rows d, num columns w, num epochs t	ConQuest [18]: Maximize epochs and minimize error	maximize $t(1 - (\frac{3}{w})^{\alpha})$
Ring buffer	Buffer length b	Netseer [87]: Maximize buffer capacity	maximize b

Table 3.1: Symbolic values and objective functions for Zipfian distributed traffic with (constant) parameter α.

Conversely, if the CMS is too small, it cannot accurately measure which keys are popular and should be stored in the cache.

To capture the balance between data structures, a programmer can combine the objectives of each data structure into a weighted sum. For the cache application, this means creating an objective function that slightly prioritizes the hit rate of the key-value store over the error of the CMS:

```
maximize 0.8*kvObj-0.2*cmsObj
```

Table 3.1 presents the symbolic values and possible objective functions for different data structures in P4All. Each structure has symbolic values and an objective function derived from the purpose of the structure, which may vary across applications. The programmer can define the objective function of each structure based on the specific needs of the system. Existing analyses of common data structures can assist in defining these functions. For example, for the Bloom filter, the probabil-

ity for false positives in Zipfian-distributed traffic has been analyzed by Cohen and Matias [21].

Some objective functions (*e.g.,* CMS) may only include a single symbolic variable, while others are a function of multiple variables (*e.g.,* Bloom filter in Figure 3.1). Because our optimizer uses Gurobi in the backend to solve optimization problems, it is bound by Gurobi's constraints. In particular, Gurobi cannot solve complex, non-linear objectives that are functions of multiple variables directly. As a consequence, we tackle these objectives in two steps. First, we transform objectives in multiple variables (say, x and y) into objectives in a single variable (say x), by choosing a set of possible values of y to consider. We create a different Gurobi instance for each value of y, solve all the instances independently (a highly parallelizable task) and find the global optimum afterwards. Second, we use Gurobi to implement piecewise linear approximations of the non-linear functions. Both of these steps benefit from some user input, and we have extended P4All to accommodate such input (described below).

To reduce objectives with multiple variables to a single variable, we allow users to provide a set of points at which to consider evaluating certain symbolic values. Doing so provides users some control over the number of Gurobi instances generated and hence the compilation costs of solving complex optimization problems. Such sets can be generated via "range notation" (optionally including a stride, not shown here). For example, a possible objective function for a Bloom filter depends on the number of bits in the filter as well as the number of hash functions used. To eliminate the second variable from the subsequent optimization objective, a programmer can define the symbolic variable `hashes` as follows.

```
symbolic hashes [1..10]
```

On processing such a declaration, the optimizer generates ten separate optimization problems, one for each potential value of the hash functions. The optimizer chooses

the solution from the instance that generated the optimal objective, and it outputs the program layout and the concrete values for the number of hashes and number of bits in the filter.

To reduce non-linear functions to linear ones, piecewise linear approximations are used. By default, the optimizer will use the simplest such approximation: a single line. Doing so results in fast compile times, but can lead to suboptimal solutions. To improve the quality of solution, we allow programmers to specify the number of linear pieces using a "step" annotation on their objective function. For instance, the objective function for cache hit rate (lines 35-37 of Figure 3.7) can be appended with a "step" of 100, indicating that a linear component is created between every 100th value.

```
objective kvsObj {
    function: sum(map(lambda y: 1.0/y,range(1,cacheEntries+1)));
    step: 100; }
```

Increasing the number of linear components in the approximation can increase the cost of solving these optimization problems. By providing programmers with optional control, we support a "pay-as-you-go" model that allows programmers to trade compile time for precision if they so choose.

In practice, we have found that non-linear optimization functions that use division can generate low-quality solutions, perhaps due to rounding errors (at least for the solver, Gurobi, that we use). Hence, we *scale* such functions up, which results in the following optimization function.

```
scale(sum(map(lambda y: 1.0/y,range(1,cacheEntries+1))))
```

3.3 Optimizer

Inputs to the P4All optimizer include a P4All program and a specification of the target's resources. The optimizer outputs a P4 program with a concrete assignment for each symbolic value, and a mapping of P4 program elements to stages in the target's pipeline. The output program is a *valid instance* of the input when the concrete values chosen to replace symbolic ones satisfy the user constraints (*i.e.,* `assume` statements) as well as the constraints of the PISA model that is targeted. In addition, loops are unrolled as indicated given the chosen concrete values. The output program is an *optimal instance*, when in addition to being valid, it optimizes the given objective function.

The P4All optimizer first analyzes the control and data dependencies between actions in the program to compute an *upper bound* on the number of times each loop can be unrolled without exhausting the target's resources (3.3.1). For example, a for-loop with a dependency across successive iterations cannot run more times than the number of pipeline stages (S) as dependent actions must be in separate stages. The unrolled program also cannot require more ALUs than exist on the target ($(F+L)*S$), where F is the number of stateful ALUs, and L is the number of stateless ALUs.

Next, the optimizer generates an integer linear program (ILP) with variables and constraints that govern the quantity and placement of actions, registers, and metadata relative to the target constraints (3.3.2). The upper bound ensures this ILP is "large enough" to consider all possible placements of program elements that can maximize the use of resources. However, the ILP is more accurate than the coarse unrolling approximation we use. Hence, it may generate a solution that excludes some of the unrolled iterations—some of the later iterations may ultimately not "fit" in the data plane or may not optimize the user's preferred objective function when other constraints are accounted for. The resulting ILP solution is a layout of the program on the target, including the stage placement and memory allocation, and optimal

concrete assignments for the symbolic values. Throughout this section, we use the CMS, as part of the data-plane cache program in Figure 3.7, as a running example. For the sake of the example, we assume that the target has three pipeline stages ($S = 3$), 2048b memory per stage ($M = 2048$), two stateful and two stateless ALUs per stage ($F = L = 2$), and 4096 bits of PHV ($P = 4096$).

3.3.1 Upper Bounds for Loop Unrolling

In its first stage, the P4All optimizer finds upper bounds for symbolic values bounding the input program's loops. To find an upper bound for a symbolic value v governing the number of iterations of some loop, the optimizer first identifies all of the loops bounded by v. It then generates a graph G_v that captures the dependencies between the actions in each iteration of each loop and between successive iterations. It uses the information represented in G_v and the target's resource constraints to compute the upper bound.

3.3.1.1 Determining Dependencies

When a loop is unrolled K times, it is replaced by K repetitions of the code in its body such that in repetition i, each action a in the original body of the loop is renamed to a_i. The optimizer constructs the dependency graph G_v based on the actions in the unrolled bodies of for-loops bounded by v. Each node n in the dependency graph G_v represents a set A_n of actions that access the same register and thus *must* be placed in the same stage.

Dependency graphs can have (1) *precedence edges*, which are one-way, directed edges, and (2) *exclusion edges*, which are bidirectional. There is a precedence edge from node n_1 to node n_2 (indicated with the notation $n_1 \longrightarrow n_2$) if there is a data or control dependency from any of the actions represented by n_1 to any of the actions represented by n_2. The presence of the edge $n_1 \longrightarrow n_2$ forces all actions associated

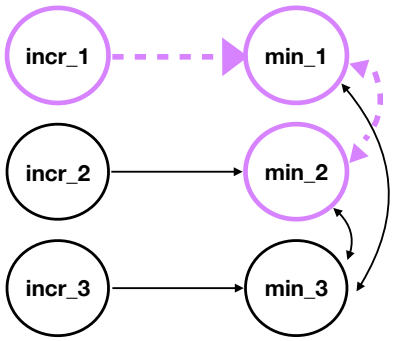


Figure 3.1: An example dependency graph used for computing upper bounds for loop unrolling.

with n_1 to be placed in a stage that strictly precedes the stage where actions of n_2 are placed. In contrast, an exclusion edge ($n_1 \longleftrightarrow n_2$) indicates the actions of n_1 must be placed in a separate stage from the actions of n_2 but n_1 need not precede n_2. In general, when actions are commutative, but cannot share a stage, they will be separated by exclusion edges. For instance, if actions a_1 and and a_2 both add one to the same metadata field, they cannot be placed in the same stage, but they commute: a_1 may precede a_2 or a_2 may precede a_1.

Figure 3.1 shows the dependency graph for `rows` from our CMS example. Only the `incr` actions access register arrays, and they all access different arrays. Thus, each node represents only one action. There is a precedence edge from `incr_i` to `min_i` as the former writes to the same metadata variable read by the latter. Thus, `incr_i` must be placed in a stage preceding `min_i`. There are exclusion edges between each pair of `min_i` and `min_j` because they are commutative but write to the same metadata fields: `min_i` sets the metadata variable tracking the global minimum `meta.min` to the minimum of its current value and the ith row of the CMS (`meta.count[i]`).

3.3.1.2 Computing the Upper Bound

To compute an upper bound for loops guarded by v, our optimizer unrolls for-loops bounded by v for increasing values of K, generating a graph G_v until one of the following two criteria are satisfied:

1. the length of the longest simple path in G_v exceeds the total number of stages S, or
2. the total number of ALUs required to implement actions across all nodes in G_v exceeds the total number of ALUs on the target (*i.e.,* $(F + L) * S$).

Once either of the above criteria are satisfied, the optimizer can use the current value of K, *i.e.,* the number of times the loops have been unrolled, as an upper bound for v. This is because any simple path in G_v represents a sequence of actions that must be laid out in disjoint stages. Hence, a simple path longer than the total number of stages cannot be implemented on the switch (*i.e.,* criteria 1). Likewise, the switch has only $(F + L) * S$ ALUs and a computation that requires more cannot be implemented (*i.e.,* criteria 2).

Figure 3.1 presents an analysis of a CMS loop bounded by `rows`. Notice that the length of the longest simple path in G_{rows} will exceed the number of stages ($S = 3$) when three iterations of the loop have been unrolled. On the other hand, when only two iterations of the loop are unrolled, the longest simple path has length 3 and will fit. Thus, the optimizer computes 2 as the upper bound for this loop.

3.3.1.3 Nested Loops

To manage nested loops, we apply the algorithm described above to each loop, making the most conservative assumption about the other loops. For instance, suppose the program has a loop with nesting depth 2 in which the outer loop bounded by v_{out} and the inner loop is bounded by v_{in}. Assume also the valid range of values for both

v_{in} and v_{out} is $(1, \infty]$. The optimizer sets v_{in} to one, unrolls the inner loop, and computes an upper bound for v_{out} as described above. Next, the optimizer sets v_{out} to one, unrolls the outer loop, and proceeds to compute the upper bound for v_{in} as described above. In theory, heavily nested loops could lead to an explosion in the complexity of our algorithm, but in practice, we have not found nested loops common or problematic. Only the SketchLearn [38] application requires nested loops and the nesting depth is just 2, which is easily handled by our system.

3.3.2 Optimizing Resource Constraints

After unrolling loops, the optimizer has a loop-free program it can use to generate an integer linear program (ILP) to optimize. Table 3.2 summarizes the ILP variables and constraints. Below, we use the notation $\#k$ to refer to the ILP constraint or variable labeled k in Table 3.2.

Action Variables. To control placement of actions, the optimizer generates a set of ILP variables named $x_{a_i,s}$ (#1). The variable $x_{a_i,s}$ is 1 when the action a_i appears in stage s of the pipeline and is 0 otherwise. For instance, in the count-min sketch, there are two actions (`incr` and `min`). If we unroll a loop containing those actions twice and there are three stages in the pipeline, we generate the following action variable set.

$$\{x_{a_i,s} \quad | \quad a \in \{\texttt{incr}, \texttt{min}\}, \quad 1 \leq i \leq 2, \quad 0 \leq s < S\}$$

Register Variables. In a PISA switch, any register accessed by an action must be placed within the same stage. Thus placement (and size) of register arrays interact with placement of actions. For each register array r and pipeline stage s, the ILP variable $m_{r,s}$ contains the amount of memory used to represent r in stage s (#2). This value will be zero in any stage that does not contain r and its associated actions. For

Variables	
Actions	#1 $\{x_{a_i,s} \mid 0 \leq s < S\}$
Registers	#2 $\{m_{r_i,s} \mid 0 \leq s < S\}$
Match-Action Tables	#3 $\{tm_{t_i,s} \mid 0 \leq s < S\}$
Metadata	#4 $\{d_i \mid i \leq U_v\}$
Constraints	
Dependencies	
Same-Stage	#5 $x_{a_i,s} = x_{b_i,s} s < S$
Exclusion	#6 $x_{a_i,s} \leq 1 - x_{b_i,s}$ $s < S$
Precedence	#7 $x_{b_i,y} \leq 1 - x_{a_i,z}$ $y, z < S, y \leq z$
Conditional	#8 $\sum_{0 \leq s < S} x_{a_i,s} = \sum_{0 \leq s < S} x_{b_i,s}$ $0 \leq i \leq U_v$
Resources	
Memory	#9 $\sum_i m_{r_i,s} \cdot w_{r_i} \leq M \quad \forall s < S$
	#10 $m_{r_i,s} \leq x_{a_i,s} \cdot M \quad 0 \leq s < S$
	#11 $m_{r_i,s} \cdot w_0 = m_{0,s} \cdot w_{r_i}$ $\forall s < S, r \geq 1$
TCAM	#12 $\sum_i tm_{t_i,s} \cdot tw_{t_i} \leq T \quad \forall s < S$
Stateful ALUs	#13 $\sum_i H_f(a_i) \cdot x_{a_i,s} \leq F$ $\forall 0 \leq s < S$
Stateless ALUs	#14 $\sum_i H_l(a_i) \cdot x_{a_i,s} \leq L$ $\forall 0 \leq s < S$
PHV	#15 $\sum_i d_i \cdot bits_d \leq P - P_{fixed}$
	#16 $d_i = \sum_{0 \leq s < S} x_{a_i,s}$ if accesses(a,d)
Hash Functions	#17 $\sum_i h_{ha_i,s} \leq N \quad \forall s < S$
Others	
At Most Once	#18 $\sum_{0 \leq s < S} x_{a_i,s} \leq 1$
Inelastic Actions	#19 $\sum_{0 \leq s < S} x_{a_{ne},s} = 1$

Table 3.2: ILP summary.

instance, to allocate the `cms` registers, the optimizer uses:

$$\{m_{\texttt{cms}_i,s} \,|\, 1 \leq i \leq 2,\, 0 \leq s < S\}$$

Match-Action Table Variables. These variables represent the resources used by match-action tables. Similar to register variables, the variable $tm_{t_i,s}$ represents the amount of TCAM used by table t_i in stage s (#3). Note that in our current ILP, we assume that all tables, ones with and without ternary matches, use TCAM.

Metadata Variables. The amount of metadata needed is also governed by symbolic values. If U_v is the upper bound on the symbolic value that governs the size

of a metadata array, then the optimizer generates a set of metadata variables d_i for $1 \leq i \leq U_v$ (#4). Each such variable will have value 1 in the ILP solution if that chunk of metadata is required and constraints described later will bound the total metadata to ensure it does not exceed the target size limits. In our running example, the bound U_v corresponds to the number of iterations of the loop that finds the global minimum value in the CMS.

Dependency Constraints. If a set of actions use the same register, they must be placed on the same stage. To do so, the optimizer adds a *same-stage constraint* (#5). Similarly, if an action has a data or control dependency on another action, the two must be placed in separate stages. If there is an exclusion edge between actions a_i and b_i, the optimizer creates a constraint to prevent these actions from being placed in the same stage (#6). If there is a precedence edge between actions a_i and b_i, the optimizer creates a constraint forcing a_i to be placed in a stage before b_i (#7).

Conditional Constraints. In some cases, as it happens in our CMS example, multiple loops are governed by the same symbolic values. Hence, iterations of one loop (and the corresponding actions/metadata) exist if and only if the corresponding iterations of the other loop exist. Moreover, if any action within a loop iteration cannot fit in the data plane, then the entire loop iteration should not be instantiated at all. Conditional constraints (#8) enforce these invariants.

Resource Constraints. We generate ILP constraints for each of the resources listed in 1.3.1. Our ILP constraints reflect the memory limit per stage (#9) and the fact that memory and corresponding actions must be co-located (#10). The optimizer also generates constraints to ensure that each register array in an array of register arrays has the same size (#11). Moreover, the ILP includes a constraint to guarantee that the TCAM tables in a stage fit within a stage's resources (#12).

To enforce limits on the number of stateful and stateless ALUs used in each stage, we assume that the target provides two functions $H_f(a_i)$ and $H_l(a_i)$ as part of the

target specification. These functions specify the number of stateful and stateless ALUs, respectively, required to implement a given action a_i on the target. Given that information, the optimizer generates constraints to ensure that the total number of ALUs used by actions in the same stage do not exceed the available ALUs in a stage (#13, #14).

To track the use of PHV, constraint #15 ensures d_i is 1 whenever the action a_i (which accesses data d_i) is used in loop iteration i. To limit the total number of PHV bits, constraint #16 sums the size in bits ($bits_d$) of the metadata d associated with iteration i and enforces it to be within the PHV bits available to symbolic program components ($P - P_{fixed}$, where P_{fixed} is the amount of metadata not present in symbolic arrays). Finally, each stage in the PISA pipeline can perform a limited number of hash functions. To capture that, the optimizer generates constraint #17, which ensures that the number of actions including a hash function h in each stage does not exceed the available number of available hashing units N.

Other Constraints. The optimizer generates a constraint so that each action a_i is placed at most once (#18). Moreover, the optimizer ensures that each *inelastic* action a_{ne} (*i.e.,* an action not encapsulated in a loop bounded by a symbolic value) must be placed in the pipeline (#19). Finally, any assume statements appearing in the P4All program are included in the ILP.

3.3.3 Limitations

Our current ILP formulation assumes each register array and match-action table can be placed in at most one stage. However, a PISA target could conceivably spread a single array or table across multiple pipeline stages. To accommodate multi-stage arrays or tables, we can relax the ILP constraint on placing actions in at most one stage (#18).

Moreover, some optimizers further optimize the use of the PHV. For example, after a metadata field has been accessed, the PHV segment storing that field could be overwritten in later stages if the metadata were never accessed again. Our prototype does not yet capture PHV field reuse.

P4All optimizes with mostly static criteria. We do not consider any dynamic components, unless a programmer incorporates a workload-dependent parameter in their objective function. P4All also does not support symbolic-width fields or parameterized packet recirculation. We leave these features, as well as PHV reuse, for future work.

3.4 Evaluation

To evaluate P4All, we implement a prototype optimizer, written in Python.

Target specification. We create a target specification for the Intel Tofino switch, based on product documentation. The specification captures the parameters in 1.3.1 and the H_f and H_l functions that specify the number of ALUs required to implement a given action. Since the Tofino design is proprietary, our specification unquestionably omits some low-level constraints not described in the documentation; with knowledge of such constraints, we could augment our target specification and optimization framework to handle them.

Compute upper bounds for symbolic values. To compute upper bounds and unroll loops, our prototype must analyze P4 dependencies. To facilitate this, we use the Lark toolkit [45] for parsing. We have also written a Python program that finds dependencies between actions and tables and outputs the information in a format our ILP can ingest. At the moment, we only produce precedence edges. As a result, we do not process exclusion edges, treating all edges as precedence edges.

Generate and solve ILP. Our prototype generates the ILP with variables and constraints in Figure 3.2, as well as the objective function. We then invoke the Gurobi Optimizer [30] to compute a concrete assignment for each symbolic value. We then use these values to generate the unrolled P4 code.

P4 compiler. After the optimizer converts the P4All program into a P4 program, we invoke the (black box) Tofino compiler to compile the P4 program for execution on the underlying Tofino switch. If our experiments initially fail to compile to the Tofino switch because of proprietary constraints, we adjust our target specification and add `assume` statements to further constrain the resource usage. Ideally, the P4All optimizer would be embedded within a target-specific compiler to automatically incorporate the proprietary constraints, without our needing to infer them.

3.4.1 Language Evaluation

When evaluating the language extension of P4All, we specifically investigate its ability to reduce code repetition as compared to P4. We implement a variety of applications in both P4 and P4All, and compare the resulting lines of code (Table 3.3).

We see reduction in lines of code with P4All because of symbolic loops that eliminate repeated code. The lines of code were reduced by 27% on average, for the applications in which the lines of code were affected. For applications that do not use any loops, P4All did not reduce the lines of code. The symbolic values in these applications often represented the number of registers in an array. In the hash table, for example, the application uses a single register array, whose size is dictated by a symbolic. There is a single set of actions for the array, that remains the same regardless of the array size, so no loops are needed. In these cases, the greater benefit of P4All is not the elimination of repetition, but rather the automated optimization of the symbolic values.

Applications	P4 Lines of Code	P4All Lines of Code
IPv4 Forwarding + Stateful Firewall (SFW [70])	282	282
BeauCoup [17]	541	541
Precision [5]	366	273
NetChain [41]	242	242
Elastic Switch.p4	804	804
Key-value store (KV)	127	127
Count-min sketch (CMS [22])	207	179
Data-plane cache (KV+CMS [42])	216	170
Non-Elastic Switch.p4 + CMS	875	847
SketchLearn [38]	445	445
ConQuest [18]	869	869
Bloom filter [9]	179	70
CMS + Bloom	318	203

Table 3.3: P4All applications, showing the lines of code in the P4 and P4All implementations.

We also note that some applications saw a greater reduction than others, due to the use of C-like macros in the P4 code. For these applications (*e.g.,* BeauCoup and ConQuest), while P4All led to a slight reduction in the lines of code in the body of the program, the addition of objective functions negated the effect. While macros make the program more compact, they make debugging more difficult. Additionally, changes to a program still require edits in multiple places, while P4All aims to make programs more robust by reducing the places a user must make edits.

3.4.2 Optimizer Evaluation

Table 3.4 reports the sizes of the constraint systems, and the compile times, for benchmark applications when compiled against our Tofino resource specification. We choose applications with a variety of features, including elastic TCAM tables (switch.p4), multivariate objectives (Bloom filter), elastic and non-elastic components (IPv4 for-

Applications	Compile Time (sec)	ILP (Var, Constr)
Linear Objective		
IPv4 Forwarding + Stateful Firewall (SFW [70])	0.4	(192, 1026)
BeauCoup [17]	0.1	(672,7511)
Precision [5]	25.7	(1316, 18969)
NetChain [41]	27.9	(252, 3278)
Elastic Switch.p4	0.2	(1080, 21581)
Non-Linear Objective		
Key-value store (KV)	15.4	(168, 857)
Count-min sketch (CMS [22])	1.8	(396, 1994)
Data-plane cache (KV+CMS [42])	27.9	(586, 2815)
Non-Elastic Switch.p4	17.5	(1498, 23575)
SketchLearn [38]	2.4	(768, 880)
ConQuest [17]	5.8	(612, 3734)
Multivariate Objective		
Bloom filter [9]	513.6 (longest) 170.0 (avg)	(240, 308) (132, 191)
CMS + Bloom	67.3 (longest) 38.1 (avg)	(658, 2266) (550, 2149)

Table 3.4: P4All applications, showing the lines of code in the P4All implementation. For structures with multiple instances, the last two columns give statistics for the single instance with the *longest* compile time and the *average* of all instances.

warding and stateful firewall), and multiple elastic components (data-plane cache, CMS and Bloom filter). In our experiments, we found that the choice of objective function greatly impacts performance. For example, a non-convex objective function results in a mixed integer program (MIP) instead of an ILP, which significantly increases solving time. On the other hand, our applications with linear objective functions (e.g., switch.p4, BeauCoup) typically had smaller compile times. Additionally, increasing the step size for an objective (i.e., reducing the number of values provided to the ILP) decreases compile time.

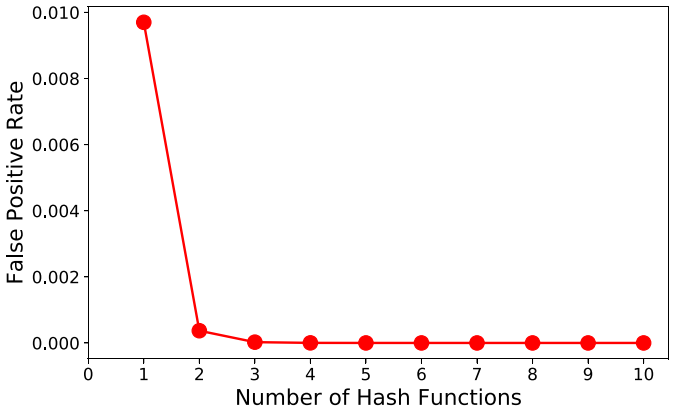


Figure 3.2: Bloom filter false positive vs. # hash functions.

For the data structures we evaluated with objective functions with multiple variables (*e.g.,* Bloom filter), our optimizer created multiple instances of the optimization problem. We report the average compile time and the average number of ILP variables and constraints *for each instance*, along with the statistics for the largest instance. Our prototype optimizer is not parallelized, but could easily be in the future, allowing us to solve many (possibly all) instances at the same time. Compile times of each ILP instance for the Bloom filter application range from roughly one second to 8.5 minutes.

In Figure 3.2, we show the objective (false positive rate) from the instances of optimization for a Bloom filter. In each instance, the optimizer increases the number of hashes used. The objective decreases for each instance, but sees diminishing returns after the first instance.

Compile time increases as we increase the number of symbolic values in a P4All program. We evaluate ILP performance by observing the solving time as we increase the number of symbolic values. Compilation for a single elastic sketch completed in about 10 seconds, while compilation for four sketches took over 30 minutes.

The number of constraints also affects compile time. The Bloom filter had the fewest ILP constraints, as it had no dependent components, and it alone had the

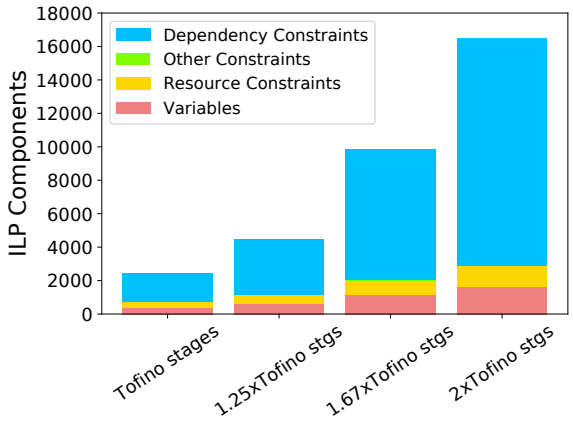


Figure 3.3: Number of ILP variables and constraints for CMS as stages increase.

largest compilation time. The reason for this is that the smaller number of constraints may lead to a more difficult optimization problem.

When we increase the available resources on the target, we generate a larger optimization problem, with more variables and constraints. Figure 3.3 shows the change in the number of constraints and variables as we increase the number of available stages on the target. Most of the resources and other constraints (*e.g.,* TCAM size, hash units, at most once, etc.) are linearly proportional to the stages. The dependency constraints are the only constraints that do not increase linearly with the stages. For a single P4All action, we create an ILP variable for each stage. However, the variables for CMS are not linearly proportional to the stages because as we increase stages, the upper bound on the actions also increases, resulting in more variables. Similarly, the ILP completion time increases super linearly with the number of stages (Table 3.5).

Some applications may have both symbolic and non-symbolic components. In our evaluations, we found that this did not significantly impact compile time. When we combined an elastic CMS (with symbolic values) and switch.p4 (with fixed-size TCAM tables), the compile time was 17 seconds. Our optimizer requires that all

Num Stages	ILP Time (s)
Tofino	1.8
1.25xTofino	4.5
1.67xTofino	53.1
2xTofino	216.0

Table 3.5: ILP completion time for CMS as stages increase.

non-symbolic portions of the program get placed on the switch, or the program will fail to compile.

Hand-written vs P4All-generated P4 To investigate whether P4All-generated P4 was competitive with hand-written P4, we examined a few P4 programs written by hand by other programmers and compared those programs with the P4 code generated from P4All. When we compare the number of registers used by the manually-written BeauCoup and the P4All-generated BeauCoup, we find they are exactly the same. ConQuest is made up of sketches, so we use the same objective function for the CMS described in 3.2: $3.0/cols$, where $cols$ is the number of columns in each sketch. With that function, our optimizer tries to allocate as many registers as possible, and allocates all available space to sketches, as more registers means lower error.

Examining the ConQuest paper in more depth, however, shows that the accuracy gains are minimal after a certain point (2048 columns). To account for this, we simply add a bound to the symbolic value for the number of columns, and as a result, the compiled code uses exactly 2048 columns as in the original. This experiment illustrates the power of P4All beautifully. On one hand, our first optimization function is highly effective—it uses up all available resources. On the other hand, when new information arrives, like the fact that empirically, there are diminishing returns beyond a certain point, we need only adjust bounds on symbolic values to reflect our new understanding of the performance. None of the implementation details need change.

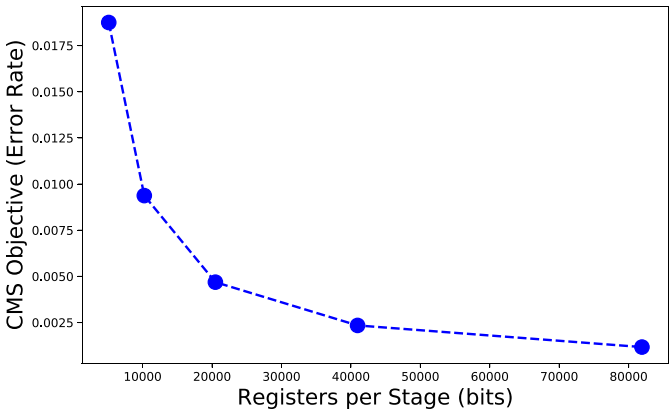


Figure 3.4: CMS error rate as memory increases.

While this analysis is admittedly ad hoc, our findings here suggest that P4All does not put programmers at a disadvantage when it comes to producing resource-efficient P4.

3.4.2.1 Elasticity

In this section, we investigate the elasticity of P4All. More specifically, we assess its ability to adjust data structures as we vary the resources available. Figure 3.4 shows how the error rate of a CMS decreases as we increase the available registers in each stage, because the P4All optimizer allocates more resources to the sketch. Figure 3.5 shows how the sizes of a KV and CMS change for different objective functions. We use the objective functions for KVS hit rate and CMS error rate as described in Figure 3.1. The first objective function $0.8 * (kvObj) - 0.2 * (cmsObj)$ gives a higher weight to the KV hit rate, while the second $0.2 * (kvObj) - 0.8 * (cmsObj)$ gives a higher weight to the CMS error rate. When we prioritize the hit rate, the key-value store gets a significantly higher proportion of resources than when we prioritize the CMS error. P4All can adapt and adjust these elastic structures to a programmer's specific requirements.

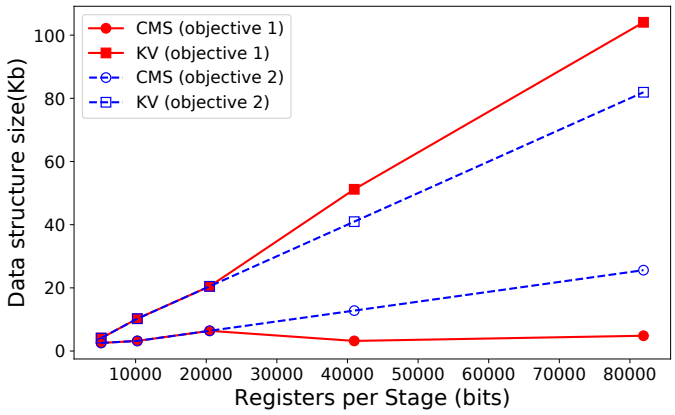


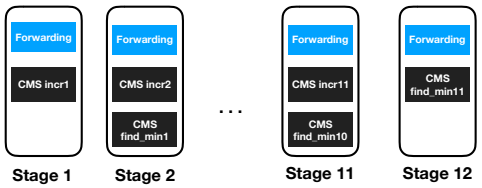
Figure 3.5: CMS and KVS sizes for different objectives.

3.4.3 Case Study

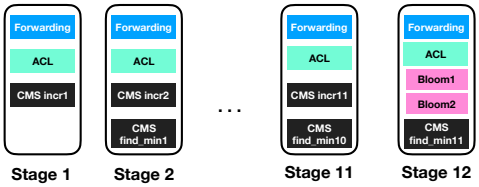
In a conversation with a major cloud provider, the researchers expressed interest in hosting a multiple applications on the same network device, which must include forwarding logic. We designed P4All for exactly such scenarios—elastic structures allow new applications to fit onto a shared device. We consider a simple case study oriented around this problem.

To do so, we started with the IPv4 forwarding code from switch.p4, but the size of the match-action table is defined symbolically in P4All. We then added a CMS for heavy hitter detection. Figure 3.6a illustrates the layout: The forwarding tables utilize all of the TCAM resources, and the CMS uses registers.

Next, to demonstrate the flexibility and modularity of our framework, we add access control lists (ACLs), which use match-action tables, and squeeze in a stateful firewall, using Bloom filters, similar to the firewall in the P4 tutorials [70]. Using P4, the programmer would manually resize the CMS and forwarding tables so the new applications could fit on the switch, but by using P4All, we do not have to change our existing code at all. To write ACLs with elastic TCAM tables, we modify the code in switch.p4 to include symbolic values for table sizes. Our optimizer automatically resizes the elastic structures to fit on the switch, resulting in the layout in Figure 3.6b.



(a) Switch layout with forwarding tables and a CMS.



(b) Switch layout with forwarding tables, ACL tables, CMS, and Bloom filter used in stateful firewall.

Figure 3.6: Switch program layouts.

The forwarding tables and ACLs now share the match-action table resources, and the registers in the Bloom filter fit alongside the CMS.

3.5 Related Work

Languages for network programming. There has been a large body of work on programming languages for software defined networks [3, 26, 61, 78] targeted towards OpenFlow [51], a predecessor to P4 [10, 57]. OpenFlow only allows for a fixed set of actions and not control over registers in the data plane, and so these abstractions are not sufficient for P4. While P4 makes it possible to create applications over a variety of hardware targets, it does not make it easy. Domino [65] and Chipmunk [28] use a high-level C-like language to aid in programming switches. P4All also aims to simplify this process, but we enhance P4 with elastic data structures. Domino and Chipmunk optimize the data-plane layout for static, fixed-sized data structures, and P4All optimizes the data structure itself to make the most effective use of resources.

Using synthesis for compiling to PISA. The Domino compiler extracts "codelets", groups of statements that must execute in the same stage. It then uses SKETCH [67] program synthesis to map a codelet to ALUs (atoms in the paper's

terminology) in each stage. If any codelet violates target constraints, the program is rejected. To improve Domino, Chipmunk [28] uses syntax-guided synthesis to perform an exhaustive search of all mappings of the program to the target. Thus, it can find mappings that are sometimes missed by Domino. Lyra [27], extends this notion to a one-big-pipeline abstraction, allowing the composition of multiple algorithms to be placed across several heterogeneous ASICs. Nevertheless, Domino, Chipmunk and Lyra map programs with fixed-size data structures, while P4All enables elastic data structures.

Compiling to RMT. Jose et al. [43] use ILPs and greedy algorithms to compile programs for RMT [11] and FlexPipe [56] architectures. These ILPs are part of an all-or-nothing compiler which attempts to place actions on a switch based on the dependencies and the sizes of match-action tables. In contrast, the P4All optimizer allows for elastic structures, which can stretch or compress according to a target's available resources.

Programmable Optimization. P^{2}GO [80] uses profile-guided optimization (*i.e.,* a sample traffic trace, not a static objective function) to reduce the resources required in a P4 program. P^{2}GO can effectively prune components that are not used in a given environment; however, if unexpected traffic turns up later, P^{2}GO may have pruned needed functionality.

3.6 Conclusions

In this chapter, we introduce the concept of *elastic data structures* that can expand to use the resources on a hardware target. Elastic switch programs are more modular than their inelastic counterparts, as elastic pieces can adjust depending on the resource needs of other components on the switch. They also are portable, as they can be recompiled for different targets.

P4All is a backwards-compatible extension of P4 that includes symbolic values, arrays, loops and objective functions. We have developed P4All code for a number of reusable modules and several applications from the recent literature. We also implement and evaluate an optimizer for P4All, demonstrating that compile times are reasonable and that auto-generated programs make efficient use of switch resources. We believe that P4All and our reusable modules will make it easier to implement and deploy a range of future data-plane applications.

While P4All effectively adds modularity to data-plane programming languages, writing applications for programmable switches usually involves more decisions than just resource allocation or data structure size. Applications may have a plethora of other parameters—timeout threshold, sampling rate, frequency of probes, to name a few. Capturing the relationship of all of these parameters with the application performance in a closed-form function is often extremely difficult, or nearly impossible. Performance may also be heavily impacted by the traffic workload, beyond what we can represent as a single constant in a P4All objective function. To this end, we need a framework capable of expressing and optimizing for more general parameters, not just data structure size, with practical objective functions.

```
1  /* Count-min sketch module */
2  symbolic rows;
3  symbolic cols;
4  symbolic cacheEntries;
5  assume cols > 0;
6  assume 0 < rows && rows <= 4;
7  struct custom_metadata_t {
8      bit<32> min;
9      bit<32>[rows] index;
10     bit<32>[rows] count; }
11 register<bit<32>>(cols)[rows] cms;
12 register<bit<32>>(cacheEntries) cache;
13 action incr()[int i] { ... }
14 action min()[int i] { ... }
15 control hash_inc( ... ) {
16     apply {
17         for (i < rows) { incr()[i]; } } }
18 control find_min( ... ) {
19     apply {
20         for (i < rows) {
21             if (meta.count[i] < meta.min) {
22                 min()[i]; } } } }
23 objective cmsObj {
24     function: 3.0/cols;
25 }
26
27 /* Key-value module */
28 symbolic k;  /* number of items */
29 assume k > 0;
30 control kv(...) {....}
31 /* Cache module */
32 control Cache( ... ) {
33     apply {
34         hash_inc.apply();
35         find_min.apply();
36         kv.apply(); } }
37 objective kvObj {
38     function: sum(map(lambda y: 1.0/y,range(
       1,cacheEntries+1)));
39 }
40 maximize 0.8*kvObj-0.2*cmsObj
```

Figure 3.7: Data-plane cache in P4All.

Chapter 4

Parasol: Automated Optimization of Parameterized Data-Plane Programs

P4All focuses on optimizing on-switch resources, including memory footprint, number of stages, or ALU usage. However, there are numerous other low-level decisions programmers must make, each of which will affect program performance. In the data-plane caching application, the allocation of memory to the key-value store and the key popularity counter is just one aspect of the program. We must also consider questions such as:

- When should we replace cached keys?
- How should we represent the counter—using a CMS, or something else?
- Is a popularity counter even the best eviction algorithm for the cache to use? Perhaps it would be better to use Precision [5], a hash table that probabilistically replaces cached keys upon collision, in place of a key-value store and CMS.

Parameter	Description
C_m	Number of columns / hashes in multi-hash table (MHT).
R_m	Number of rows (cells per hash) in multi-hash table.
C_c	Number of columns / hashes in count-min sketch (CMS).
R_c	Number of rows in CMS.
T_t	Timeout threshold for cache.
T_r	Replacement threshold.
P	Use Precision in place of MHT + CMS.

Table 4.1: Parameters of the data-plane cache.

Clearly, there are many ways to implement a cache. If we imagine a program that describes all the implementations of a cache that a programmer can imagine, then each of the design questions corresponds to a parameter in that program. Table 4.1 provides a non-exhaustive list of the parameters in a cache.

P4All requires objectives to be defined as a function of a program's parameters. Unfortunately, writing a closed-form function, that includes all of the parameters, to represent the hit rate for a data-plane cache is an arduous task. Hit rate is not easy to *predict* from the values of the parameters, let alone model analytically. The performance of the cache depends on a number of factors. A key is evicted from the cache when there is a hash collision, so the hit rate is influenced by the probability of collisions. However, not all collisions result in a key being replaced. If the key tracker is a CMS, the choice to insert an uncached key after a collision depends on the stored count for that key, and thus, the hit rate also depends on the accuracy of the counts in the CMS.

The interaction of all these factors is not straightforward—they depend on the workload distribution. Theoretical models would then have to make assumptions about that distribution [22]. *Even if* the programmer goes through the considerable effort of working out a closed-form objective function for a cache, it can only express a theoretical miss rate; the actual rate may be drastically different in practice [16].

The importance of optimizing for the expected workload can be illustrated by comparing cache performance across workloads and configurations. The hit rate of a cache depends on which keys are in the cache, which is determined not only by how large the data structures are but also the choice of that data structure (CMS vs. Precision) and the timeout threshold. Certain parameters can have a large range of potential values (*e.g.,* timeout could range from milliseconds to seconds to even longer). The subset of that range that performs well in practice can be quite small—a too-small timeout means that moderately popular keys will get frequently evicted and re-added, while a too-large timeout can result in less popular keys staying in the cache for far too long. P4All does not allow us to express parameters such as timeouts, making it easy to pick suboptimal values.

We develop Parasol as a more flexible framework, in which parameters can represent almost any aspect of the program, and objective functions express high-level performance goals as simple Python programs.

4.1 Parasol

Parasol is a novel, general framework for synthesizing data-plane programs, consisting of two parts (as in P4All): a sketching language and an optimizer. The sketching language is an extension of Lucid [68], a high-level, event-based data-plane programming language. We choose Lucid, as opposed to P4, because Lucid has a number of components (*e.g.,* a module system) that allow for more flexibility. Parasol programmers write program *sketches* [67], which are Lucid programs with symbolic values. These symbolic values are similar to those in P4All, and they represent the *parameters* of the program; each is an undefined value that will be filled in by the optimizer. Parameters in Parasol are highly flexible; they can control just about any aspect of the implementation.

The optimizer uses an iterative search algorithm to automatically optimize the parameters according to a user-defined performance objective. These objectives come in two parts. The first part measures arbitrary aspects of a program executing in simulation mode over an example traffic trace. The second part computes an arbitrary, user-defined score based on those measurements. Both parts are implemented in Python rather than the more limited languages of switch data planes. Hence, users can express essentially unlimited optimization criteria—the main constraint is the fidelity of the simulation environment to reality. The optimizer simulates the program's behavior on traffic traces drawn from a particular network, allowing more tailored optimizations than would be possible from relying solely on static, workload-independent quantities such as switch memory resources and architecture.

In summary, Parasol is a new data-plane sketching language and optimization framework with the following features:

- **Flexible objectives:** Parasol's optimization algorithm can optimize for a wide variety of *high-level* metrics such as hit rate or measurement accuracy.

- **Flexible programs:** The parameters of a Parasol program may control many properties, including probe generation frequency, algorithmic choices, memory layout, data-structure selection, or threshold values.

- **Flexible environments:** Parasol programmers may tailor their optimization to particular network environments by providing representative traffic traces.

We evaluate Parasol by fully implementing and optimizing ten different data-plane programs, with various parameters and objective functions. Our experiments found that the Parasol optimizer completed a simulation iteration in approximately eight minutes on average (with an average trace size of two million packets), and all applications could be optimized with a time budget of two hours (i.e., with fifteen iterations on average). The solutions produced by the optimizer not only complied

with hardware resource constraints, but were comparable in performance to hand-optimized P4 code.

4.2 Lucid Background

The first component of Parasol is a *sketching language* (similar to P4All), written in Lucid, that allows users to write parameterized programs. Lucid uses C-like syntax to provide an *event-based* view of the network, in which incoming packets are represented as *events*. When a packet arrives at the switch, the event's *handler* is executed. Handlers run directly on switch hardware, and may read and modify header values and register arrays, as well as drop, create, and forward packets. Lucid provides two backends: a simulation-based interpreter and a compiler to P4.

As discussed in 1.4.1, challenges of data-plane programming are not specific to P4; they appear Lucid as well. While Lucid raises the level of abstraction of P4, it is still very tightly coupled with the underlying hardware. Programmers must understand how each component will get mapped to corresponding resources, and they must hardcode parameter values accordingly.

To introduce Lucid and its challenges, we show an implementation of a data-plane cache in Lucid. The code is included at the end of this chapter in Figure 4.4.

Similar to P4, we first define the data structures used in the application: a register array for the key-value store, and register arrays for the key popularity tracker (*e.g.,* a CMS).

```
/* A register array for cache hash table */
global Array.t<32> keyValue = Array.create(1024);
/* A register array for each CMS row */
global Array.t<32> counter0 = Array.create(2048);
...
global Array.t<32> counter3 = Array.create(2048);
```

When a request appears at the switch, it triggers the `request` event, which corresponds to the `apply` block in P4 code. The lines of code within the `request` event perform the same hashing and minimum computation actions as in the P4 code (lines 31-44 in Figure 2.2). Instead of metadata, we use integer and boolean variables to store intermediate values.

```
event request(int key) {
  int cachedKey = // Retrieve key stored in cache
  bool found = (key == cachedKey); // Check if request matches cached
     key
  if (found) {generate response(); } // Respond to request if key is
     cached
  else { // Track key popularity with a CMS
    /* Compute hash indexes for each row */
    int cms_idx0 = hash(...);
    ...
    int cms_idx3 = hash(...);
    /* Retrieve and increment stored counter values in each row */
    int count0 = GetAndIncrStoredValue(counter0, cms_idx0);
    ...
    int count3 = GetAndIncrStoredValue(counter3, cms_idx0=3);
    /* Calculate min across stored counter values */
    int min = 0xffffffff; // Initialize global min
    if (count0 < min) { min = count0; }
    ...
    if (count3 < min) { min = count3; }  } }
```

We chose Lucid as the basis of our tool for two reasons. First, as a high-level language, it provides useful abstractions for representing the numerous decisions programmers must make during implementation. Second, Lucid provides an interpreter that can simulate a program's behavior without compiling it. The interpreter runs a

network-wide simulation, and can be run on different input traces, allowing the same program to be optimized for different traffic profiles with no additional user effort.

4.3 Language Extension

To implement Parasol, we add three new features to Lucid. First, we add symbolic values (à la P4All symbolic values in 3.1.1) to represent the parameters of a program that should be optimized. Second, we add a way to select between two different data structures based on a symbolic value. Finally, we add a foreign function interface that allows the user to take arbitrary measurements of the network during simulation. Figure 4.5, included at the end of this chapter, shows a pared-down example implementation of a data-plane cache that we use to demonstrate these extensions. Parts of the program that do not relate to Parasol's extensions have been omitted.

Symbolic values. Symbolic values in Parasol function as placeholders that may take on any value of the given type. Each is later replaced with a concrete value, supplied during the compilation/optimization process. Once declared, a symbolic is used in the same way as a compile-time constant. Parasol symbolic values are more general than those in P4All, and can take on any type that is supported by Lucid, but they are most commonly integers or booleans. Parasol symbolics can represent any configurable parameter in a program—data structure size, heavy hitter threshold, timeout value, sampling rate, etc. The data-plane cache in Figure 4.5 contains six symbolic values (see Table 4.1 for a list of parameters).

```
symbolic size rows;
symbolic int cols;
symbolic int cacheEntries;
symbolic bool useCms;
symbolic int timeoutThresh;
symbolic int replacementThresh;
```

We note that the `size` type is used to specify the size of integers in bits. If a symbolic value is used in a bounded loop (described in 3.1.2), it must be of type `size`.

The boolean parameter represents the selection of a data structure—if true, the cache will use a CMS to track key popularity, otherwise it will use an alternative structure, such as Precision [5]. If a CMS is used, `replacementThresh` determines the threshold for adding new keys to the cache, and `timeoutThresh` determines when keys in the cache are considered expired.

As in P4All, programmers can describe bounds on any symbolic value (in JSON format):

```
"bounds" : {
  "rows" : [1,4],
  "cols" : [32, 2048],
  "cacheEntries" : [32, 1024],
  "timeoutThresh": [500000,50000000],
  "replacementThresh" : [5000, 100000]
}
```

We can define a CMS in Parasol with a symbolic array of register arrays, similar to P4All:

```
Array.t<32>[rows] cms = [Array.create(cols) for i<rows];
```

In Parasol, we can use a bounded loop in the event handler to define actions that are repeated over each row of the CMS:

```
event request(int key) {
  int min = 0xffffffff; // Initialize global min
  for (i < rows) {
    /* Compute hash index */
    int cms_idx = hash(...);
    /* Retrieve and increment stored counter value */
    int count = GetAndIncrStoredValue(cms[i], cms_idx);
    /* Calculate min across stored counter values */
```

```
    if (count < min)
      { min = count;}
  ... } }
```

Selecting data structures. In order to write a program that may implement one of two different structures, those structures need to implement the same interface, so they can be used interchangeably in a program. Lucid provides a standard *module* system for representing data structures. Each module contains definitions for zero or more types, functions, and events. In Figure 4.5, the CMS and Precision modules both contain definitions for a type `t`—the type `CMS.t` represents a count-min-sketch structure, while the type `Precision.t` represents a Precision data structure. They also contain functions for initializing those structures, and functions for deciding when to add a particular key to the cache.

```
module CMS : {
  type t = ...;
  fun t create(size rows, int cols) = {...}
  fun int getCount(int key) {...}
  un bool decideIfAdding(int key) {...} }
module Precision : {
  type t = ...;
  fun t create(size rows, int cols) = {...}
  fun int getCount(int key) {...}
  fun bool decideIfAdding(int key) {...} }
```

Although `CMS` and `Precision` are the actual modules, they are not referenced anywhere else in the program. Instead, the rest of the program uses the `KeyTracker` module, which is an alias for either `CMS` or `Precision`, depending on the symbolic value `useCms`.

```
module KeyTracker=CMS if useCms else Precision;
global KeyTracker.t tracker = KeyTracker.create(rows,cols);
```

The program may then simply call the function `KeyTracker.create` to initialize the tracker[1], and similarly use the function `KeyTracker.DecideIfAddingKey` to determine if a key should be added to the cache.

Parasol's extension to the Lucid type checker makes sure `CMS` and `Precision` contain exactly the same declarations (*i.e.,* implement the same interface), which allows the program to use `KeyTracker` safely while remaining oblivious to implementation-level differences between `CMS` and `Precision`. If the modules differed, the programmer could instead create wrapper modules to ensure they present the same interface.

4.3.1 Objective Functions

Foreign function interface. Parasol implements a simulation-based optimization, in which an optimization system chooses values for each parameter, simulates the application over a sample traffic trace, and measures the resulting performance. Hence, our final extension to Lucid is a *foreign function interface* that lets a programmer instrument their code with calls to external measurement functions that are executed by the Parasol simulator, but removed from the final compiled program. In Figure 4.5, the extern `logHits` (defined in Figure 4.1) is a function implemented in Python by the programmer, which counts the number of cache hits and misses while the Parasol simulator is running. Each time a cache lookup is performed, `logHits` is called to record whether the lookup was a hit or a miss. After completing a simulation, the Parasol optimizer uses these measurements evaluate the program's performance.

Objective and measurement functions are often simple. For our data-plane cache, the goal is to maximize the hit rate. The functions for measuring and computing miss rate can be defined in just seven lines of Python (Figure 4.1).

The measurement function `logHits` is called from the Parasol program once per request, as in Figure 4.5. The `objective` function is called by the optimization algo-

[1]The `tracker` variable is annotated as **global** to indicate that it is a persistent structure stored in register arrays.

```
1 hits = 0
2 misses = 0
3 def logHits(found):
4   if found:  hits += 1
5   else  :  misses +=1
6 def objective():
7   return misses/(hits+misses)
```

Figure 4.1: Measurement and objective functions in Python for the data-plane cache.

rithm at the end of simulation. The global variables `hits` and `misses` are maintained in a single instance of the Python interpreter, so their values persist throughout the execution of the program.

Parasol permits only extern functions that have no return value, but does not impose any requirements on what can be passed as a parameter to these functions. Since externs also cannot modify any Lucid program state, this means they can be safely elided during compilation.

The objective can be calculated using any part of the operating environment. Examples of these objective functions include the distribution of flows across paths in a load-balancing application, the rate of collisions in a hash table, and the comparison of a CDF created from run-time measurements to a ground truth CDF. The optimizer treats the objective function as a black box; any metric used by the function is acceptable.

Comparing with ideal implementations. A particularly useful type of measurement is to compare the runtime behavior of a data structure against an idealized implementation. As an example, a data-plane application can produce round-trip time (RTT) samples by matching SYN packets with corresponding SYN-ACKs [19, 62]. When the switch sees a SYN packet, it stores its timestamp in memory, and can compute the RTT when it sees the corresponding SYN-ACK packet. However, if the structure is full, the switch cannot store new SYN packets; as a result, the

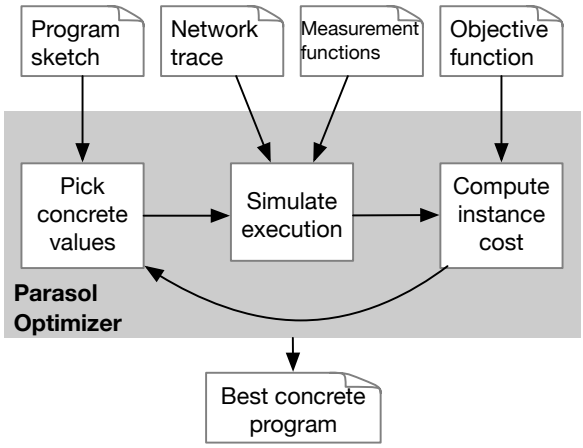


Figure 4.2: Overview of the Parasol optimization framework.

application can only provide a portion of RTT measurements. During simulation, a measurement function could maintain a Python data structure which does not run out of memory, and compare its results to those of the Parasol structure—this provides an easy-to-compute ground truth for how well the Parasol program could perform.

4.4 Optimizer

The second component of Parasol is a framework for automatically optimizing the parameter values of a program sketch; a high-level overview of this framework is provided in Figure 4.2. The programmer provides four inputs: (1) a program sketch (with symbolic values), (2) a traffic trace, (3) one or more *measurement functions*, and (4) an *objective function*. The Parasol optimizer then finds effective values for the parameters of the program using an iterative search algorithm. In each iteration, the search algorithm selects a concrete value for each symbolic value. The resulting program is then simulated on the provided traffic trace using the Lucid interpreter.

During simulation, measurements are taken via calls to the measurement functions, using Parasol's foreign function interface. At the end of simulation, the objective function uses these measurements to score the concrete program. The search algorithm then uses the historical series of those scores to select new concrete values

for the next iteration. This process repeats for a set time budget. At the end, the optimizer returns the highest-ranked concrete program that successfully compiles to the underlying hardware.

4.4.1 Simulation

We use a modified version of the Lucid interpreter to model the behavior of Parasol programs on a traffic trace. The interpreter simulates the passing of messages between one or more switches in a network, running the appropriate Lucid code when each is received. The simulation includes important switch features such as recirculation and timestamps. To enable execution of Parasol programs, we augmented the interpreter to handle symbolic values and foreign functions.

Although the Lucid interpreter models many important aspects of a network, it is not perfect. For example, it provides only a limited model of transmission delay, so properties such as packet reordering are difficult to measure accurately. However, its limitations are not fundamental; the interpreter could certainly be extended further to accommodate an even wider variety of potential applications.

4.4.2 Search Algorithm

The final component of the Parasol optimizer is the search algorithm itself. The goal of the search algorithm is to find parameter values that minimize the objective function. However, the space of possible solutions can be intractably large. Doing an exhaustive search is inefficient, and a naïve strategy may never discover a compiling solution.

As a strawman solution, Parasol could require users to define the search space by providing bounds on all variables. However, this will almost certainly include a large number of non-compiling solutions, as even experts would have trouble determining the correct bounds. As an example, reasonable bounds on cache with a CMS as the

key tracker might be 1-5 cache tables and CMS rows, and cache entries and CMS columns that are less than the amount of memory in a stage. These bounds produce a solution space of 4225 configurations, only 16% of which compiled to our target switch.

Alternatively, Parasol could use a heuristic to test if each configuration will compile before it is simulated. If the configuration does not compile, Parasol can assign it a maximum cost. While this avoids simulating non-compiling configurations, it also reduces the effectiveness of the search strategies, as it does not give any indication of a direction in which to search. One could imagine simulating anyway, in the hopes that it will lead us to a compiling configuration, but this is unlikely—programs using an impossible amount of memory, for example, are likely to perform impossibly well.

In practice, we address this issue by splitting the search algorithm into two phases: preprocessing and simulation. In the first phase, Parasol automatically prunes non-compiling solutions from the search space, without requiring user-defined bounds. In the second phase, Parasol searches the space of remaining solutions with a user-configurable search algorithm.

Preprocessing. In a nutshell, the goal of the preprocessing phase is to ensure our solutions are making maximal use of the resources on the switch, without using so many that the program fails to compile. Accordingly, during this phase we only consider symbolic values which affect resource allocation. The resources we consider are memory, pipeline stages, hash units, array accesses, and ALU usage (1.3.1. We assume that the program is monotonic with respect to resources—that is, increasing the value of any symbolic value should not decrease the amount of resources used. In our experience, this is a safe assumption; we note that all of the applications we evaluated satisfied this property.

The optimizer begins by setting all symbolic values to either a default or user-provided starting value. We then pick a symbolic, and determine an upper bound for

it by iteratively increasing only that symbolic's value until we run out of resources. Thanks to monotonicity, the largest value that fits provides an upper bound for the symbolic.

We then pick another symbolic and repeat this process; however, this time we find one upper bound for each possible value of the first symbolic. We do the same for the next symbolic, and the next, each time finding an upper bound for all valid combinations of previously-processed symbolics. When we finish, we will have enumerated the entire useful search space (*i.e.,* every compiling solution).

This process, however, grows multiplicatively with the number of parameters. To make it more tractable, we use domain knowledge to set a reasonable default starting value that allows Parasol to discover the entire useful search space, without having to compile every solution in that search space. Values that represent memory used per stage are initially set to the max memory available in a stage, and values that contribute to other resources start at 4. We choose 4 as a starting value because we found it generalized well to all of our applications, providing a significant reduction in preprocessing time when compared to a starting value of 1. For example, the preprocessing time for caching structure with a Precision key tracker improved from almost 2 hours to only 25 minutes.

Simulation. In the second phase of the search algorithm, we perform a configurable search through the pruned space of solutions we created in phase one. We choose a configuration from phase one, select values for any non-resource symbolics, and execute the resulting program in the Lucid interpreter. We then score the configuration based on its output, and use a search strategy to select the next configuration to evaluate based on the history of scores.

The Parasol optimizer does not rely on any particular search strategy; rather, it is able to accommodate a variety of search algorithms. Programmers can take advantage of this to improve the search process. Given the knowledge of a particular application

and its parameters, the programmer may know which strategies will be most efficient, and can choose those instead of the strategies we provide. We provide four built-in search functions for programmers to use—exhaustive search, Nelder-Mead simplex method, simulated annealing, and Bayesian optimization—but Parasol also supports any programmer-defined search of the solution space, and is compatible with any technique written in Python (*e.g.,* stochastic gradient descent, genetic algorithms, etc.). We choose these strategies because (with the exception of exhaustive) they use the history of scores to efficiently navigate the search space. They also provide a range from simple (exhaustive, Nelder-Mead simplex) to more complex (Bayesian). Programmers are free to choose the search algorithm that provides their preferred balance between search time and optimality of the final result. We evaluate the effectiveness of each of these strategies and analyze how the choice of strategy affects the optimizer in 4.6.

4.5 Design Tradeoffs

Accelerating preprocessing. The first phase of optimization requires analysis of the resource usage of a program to determine if it will compile. The simplest way to do this would be to actually compile the program; however, compilation can be very slow (the ConQuest [18] application took over 13 minutes), and most applications require compiling many configurations (ConQuest has a compiling search space of 25 configurations). Instead, we have tested a range of heuristics, with varying trade-offs between performance and accuracy.

All three of our heuristics operate by attempting to assign each action in the Lucid program to a stage of the switch's pipeline. The primary distinction between the heuristics is the types of resources they account for during placement. Our simplest heuristic, dataflow graph, only accounts for dependencies between actions (two actions

Heuristic	Avg compile time	Reduction
Dataflow graph	51s	–
Greedy layout	51s	13%
Lucid-P4	1.5min	13%
Full compilation	1.5min	16%

Table 4.2: The performance of preprocessing heuristics for a single configuration, averaged over each evaluated application. The greedy layout provides the best balance between performance and accuracy.

cannot be in the same stage if one depends on the output of the other). Our next heuristic, greedy layout, additionally considers the layout of memory, hash units, array accesses, and ALU usage (for example, we cannot have multiple concurrent accesses to the same array). Our final heuristic is to run a partial compilation—rather than fully compiling to the switch, we instead compile Lucid to P4. This is much faster than a full compilation, and additionally considers resource limits on physical tables in the pipeline (such as match column width, maximum table size, and number of actions per stage). We note that heuristics can only underestimate, never overestimate, resource usage. In other words, the solutions that do not compile with a heuristic will also never compile to the target device.

The only constraints that we encountered which were not modeled by the Lucid compiler are packet header vector (PHV) clustering constraints—each packet header or metadata variable in a program must be placed into a specific PHV cluster, and each cluster has a fixed number of ALUs in each pipeline stage. In our experience, it was possible to run afoul of PHV constraints in sufficiently complicated programs, but these violations were unaffected by choice of parameter values. Our preliminary implementations of 6/10 applications failed to compile with *any* configuration due to PHV constraints, but once we adjusted the programs to accommodate for the constraints, we did not run into PHV constraint violations for any configurations.

A summary of the performance of our heuristics appears in Table 4.2. We list the average compile time for each of our evaluated applications and the average reduction in search space size, using the dataflow graph heuristic as the baseline. In practice, we have found that the greedy layout heuristic provides the best trade-off between performance and accuracy. We cope with the potential inaccuracy of the heuristic by including a safeguard to ensure that Parasol returns a compiling solution. Specifically, we actually compile the highest-ranked configuration at the end of our optimization loop. Should compilation fail, Parasol tries the next-highest-ranked, and so on, until one compiles. If none of the tested solutions compile, the system will repeat the optimization process, excluding solutions that did not compile.

We found that in practice, this rarely happens. After manually fixing any PHV errors, the optimal solutions for nine out of the ten applications fit within the target resources. One of the applications (CMS) resulted in "optimal" configurations that did not compile. However, the Parasol optimizer found a compiling solution that had similar performance.

Unrepresentative traces. Since the Parasol optimization framework is simulation-based, it relies on a representative traffic trace. If the actual traffic in the network deviates from the patterns in the trace, the performance of the application may not match the simulated performance. However, because Parasol preserves the semantics of the data-plane program, it will never produce unexpected or invalid behavior—its performance may simply be poorer than anticipated.

To mitigate poor performance, programmers can use multiple traffic traces to optimize their application, and use a weighted combination of performance on the traces as the objective function. We show an example with our data-plane cache in 4.6.3.1, by optimizing with workloads of different distributions. Alternatively, if the distribution depends on time of day, the programmer can use traces from peak times, where applications are likely most sensitive to poor performance.

Beyond poor performance, an unrepresentative trace can leave an application vulnerable to attacks when the training trace only contains benign traffic. To use Parasol for tuning a security system, one needs traces containing the kinds of attacks the application seeks to detect or prevent. Fortunately, Parasol users need not acquire and label such traces themselves, as the network security community already goes to great lengths to produce and share traces for the evaluation of their own security systems [8]. These traces come from a variety of sources, including cyber defense exercises [14] and security-oriented testbeds or simulators [15, 77].

4.6 Evaluation

Our evaluation of Parasol addresses its two components:

- **Language.** Can Parasol express a wide variety of parameters, objective functions, and data-plane applications?
- **Optimizer.** How well do optimized Parasol programs perform, and how quickly does Parasol find good parameters?

To answer these questions, we used Parasol to implement and optimize a suite of ten data-plane applications with respect to representative traffic traces. We chose applications that encompass a wide array of structures (including commonly used structures like sketches and hash tables) and contain a diverse set of parameters and objective functions. Our application and optimizer code is publicly available. [2]

In the remainder of this section, we discuss each Parasol component individually, and finish with two in-depth case studies. We used three types of traces in our evaluation—the University of Wisconsin Data Center Measurement trace [7], a trace from core Internet routers [13], and synthetic traces for the cache application. Unless

[2] https://github.com/mhogan26/Parasol

	Classes of parameters in application					P4All?	
Application	**mem. alloc.**	**threshold**	**data struct. choice**	**timing**	**Objective (LoC)**	**Params**	**Obj.**
Count-min sketch (CMS)	√				Mean estimate Error (20)	√	√
Multi-hash table (MHT)	√				Collision ratio (11)	√	√
Data plane cache (KV [42, 71])	√	√	√	√	Miss rate (23)	✗	✗
RTT monitor (RTT [19])	√			√	Read success rate (118)	✗	✗
Unbiased RTT (Fridge [86])	√	√			Max percentile error (88)	✗	√
Starflow [69]	√				Eviction ratio (17)	√	✗
Conquest [18]	√				F-score (101)	√	✗
Load balancing (LB [75])	√	√			Error vs. optimal (38)	✗	✗
Precision [5]	√				Avg. error for top flows (28)	√	✗
Stateful Firewall (SFW [68])	√	√		√	Packet overhead (70)	✗	✗

Table 4.3: Applications optimized with Parasol, showing which classes of parameters/objective functions were used, and which of them could be expressed in P4All.

otherwise stated, we split a single input trace into a training trace and testing trace (see Figure 4.4 for trace sizes).

4.6.1 Language Evaluation

To evaluate the expressiveness of Parasol, we implemented applications with multiple classes of parameters and diverse objectives. The right two columns of Table 4.3 show the high-level benefit of Parasol over P4All: whereas Parasol allowed us to fully express the optimization goal of each application (parameters and objective function), P4All could only express the full optimization goals of 2/10 applications. In the rest of this section, we discuss the ability of Parasol to represent a diversity of both parameters (its "program flex"), and objective functions.

Program flex. As Table 4.3 shows, the Parasol programs we implemented had four general classes of parameters: memory allocation, decision thresholds, choice of data structure, and operation timing. These classes encompassed a diverse range of parameters, including data structure size and the probability of an item being added to a structure. Parasol's flexible approach allowed it to handle all of them. In comparison, P4All could only support parameters from 6/10 of our implemented applications (CMS, MHT, Starflow, Conquest, Precision) as it is impossible to express threshold, timing, or data structure choice parameters in P4All.

Even for the examples that *could* potentially be optimized by P4All, it is easy to imagine slightly more complex variants that would require incompatible parameters. For example, our CMS is a simple implementation with no concept of time intervals—it never resets. Most applications, however, will want to count over intervals, which requires a mechanism to periodically reset or age counters, and a parameter that controls the length of the interval. The addition of that one simple parameter makes the "deployable" variant of CMS incompatible with P4All.

Objective functions. The objective functions for our applications measured a wide variety of high-level properties (Figure 4.3). These functions were generally short and simple: on average, each function was approximately 50 lines of Python code. The only requirement for Parasol objective functions is that they be expressible in Python. They can include any, all, or none of the parameters in the application, along with any measurements taken during the simulation.

In contrast, P4All requires programmers to supply a closed-form objective function, which specifies exactly how the parameters relate to the final cost. In practice, this can be very difficult, particularly for applications that do not have theoretical guidelines or proven error bounds. This is common, even in research, where many data-plane applications are evaluated empirically, without finding provable theoretical guarantees [18, 69]. Furthermore, many systems are composed of multiple components or data structures; writing a closed-form function for those systems requires not just understanding each component individually, but codifying precisely how they interact.

In our evaluation, we considered an objective function to be expressible in P4All only if we could find a derivation in existing literature. We consider deriving a closed-form objective function to be beyond the scope of an application developer (and also this paper) as it requires significant theoretical work. We required that functions include all the parameters of the applications, but did not require those parameters to

be expressible in P4All. Although functions needed not be for a single component, we note that none of our applications with multiple structures (KV, Starflow, Conquest) had a closed-form function.

With these criteria, we were only able to express three out of our ten objective functions in P4All. Even so, there is a caveat: functions from the literature typically quantify worst-case performance. These objective functions oftentimes do not provide a realistic idea of how the application performs in practice. In contrast, Parasol objective functions measure actual performance on a sample trace, and are therefore able to optimize for a much broader range of criteria, even when a closed-form error function exists [16, 54, 88]. We compare Parasol against a closed-form objective for the unbiased RTT (Fridge) application in detail in §4.6.3.2.

We note that while were able to express a simple version of the data-plane cache in P4All, we cannot implement the cache with all of the parameters listed in Table 4.1. Additionally, the P4All objective function is a weighted sum of the theoretical hit rate and error rate of the key-value store and CMS, respectively, but it does not capture the relationship between the data structures—the accuracy of the CMS will affect the hit rate of the cache, because it can more effectively identify popular keys. Hence, we cannot fully express the data-plane caching application in P4All.

In general, P4All works well if programmers want to bound worst-case performance for an application. However, that is not often the case in practice; programmers typically develop applications empirically, without deriving error bounds. Parasol takes the ideas of P4All and tailors them to a more practical scenario, which can be applied to nearly any data-plane application, regardless of if error bounds exist.

4.6.2 Optimizer Evaluation

When evaluating the Parasol optimizer, we investigate how well optimized Parasol programs perform, and how quickly Parasol finds good parameters for those programs.

We also detail two case studies (using the data-plane cache and Fridge [86] data structures) that illustrate the practical benefits of using Parasol.

4.6.2.1 Optimization Quality

We evaluate the quality of Parasol's solutions, compared to both hand-optimized systems and an oracle optimizer (described below) and analyze the factors that impact it. All experiments in this section are based on a two-hour time limit for the dynamic search phase of the Parasol optimizer.

First, we compare the results of optimization with Parasol to optimization with an "oracle". Whereas the Parasol optimizer chooses parameters on a training data set, separate from the testing data, the oracle optimizer chooses parameters by exhaustively searching the testing data set, *i.e.,* it always chooses the optimal parameters.

Parasol found configurations that performed as well as the oracle for 6/10 applications (CMS, MHT, RTT, Starflow, Precision, and SFW). For 3/10 applications (KV, Fridge, ConQuest), the relative difference between the objective score of Parasol's and the oracle's configuration was under 12% (*i.e.,* $\frac{|\text{Objective}_{\text{oracle}} - \text{Objective}_{\text{Parasol}}|}{\text{Objective}_{\text{oracle}}}$). For the remaining application, the load balancer (LB), Parasol's solution was, in relative terms, 82% worse than the oracle. However, in absolute terms the difference was small: the oracle's configuration performed 1.7% worse than a perfect load balancer, while Parasol's configuration performed 3.1% worse than a perfect load balancer.

The Parasol preprocessor. To measure the effect that Parasol's preprocessor had on the solution quality, we compared application performance when optimized with and without preprocessing, using the same two-hour time budget for Parasol's search phase. When the preprocessor was disabled, we bounded the search space by setting the same initial bounds for all memory allocation variables—20 register arrays (*e.g.,* cache tables) and the max amount of SRAM per stage for registers (*e.g.,* cache entries per table). In our judgement, this represented a reasonable bound—high

enough to include all compiling solutions for each application without unnecessarily inflating the search space. Additionally, without the preprocessor, we assigned a predetermined max cost to solutions that did not compile (*e.g.,* 100% cache miss rate), to avoid simulating them.

Preprocessing consistently improved application performance, especially for applications that had a large search space or used multiple structures that compete for resources (Starflow, KV, ConQuest, SFW). In fact, when the cache used CMS as the key tracker, Parasol consistently did not find a compiling solution in the time budget without preprocessing.

- For ConQuest, enabling the preprocessor improved recall from 75% to 87%.
- For Starflow, the preprocessor improved eviction ratio from 35% to 15%.
- For the stateful firewall, the preprocessor improved recirculation and retransmission overhead from 16 kbps to 0.01 kbps.

Applications that had a small search space (CMS, MHT, Fridge, LB) did not perform significantly better when preprocessing was enabled. However, even for such applications, preprocessing still has an important benefit: it automatically bounds the search space for the programmer, without the need for them to manually "guess" reasonable bounds.

The Parasol searcher. We found that the effectiveness of Parasol's search phase depended on two factors: the search strategy and the quality of the input trace. Parasol provides four built-in strategies: exhaustive search, Bayesian, simulated annealing, and Nelder-Mead simplex. We note that all of these strategies (except exhaustive) have hyperparameters that control the learning process. We chose hyperparameter values manually such that strategies produce solutions as good as or near the oracle solutions. We found that we could re-use these values for all applications without negatively affecting solution quality.

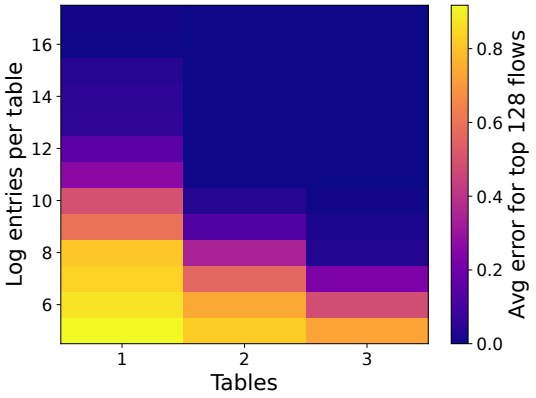


Figure 4.3: Average error for top 128 flows in the Precision application for different configurations. A darker color represents a lower error. The optimal configuration achieved an error of 0.01%, and nearly 40% of the solution space produced an error of less than 1%.

Search strategy. For some applications, the choice of search strategy does not matter because a large portion of the compiling solution space is near-optimal. For example, in the Precision application, over half of the search space after preprocessing contained solutions that produced an average error of less than 10% (Figure 4.3), compared to the optimal of less than 1%. In such cases, the search methods mostly converged to the same configuration or to configurations that had very similar performance.

For more complex applications, we found that no single search strategy dominated. Because of this, we found that the best strategy was to run multiple strategies in parallel for each application, and choose the best result from among them. Conversely, for applications with a small search space (after preprocessing), we simply used exhaustive search. We consider a search space to be small if the exhaustive search completed within the two-hour time budget.

Training trace. Across all applications, we found that traces with approximately 1 million packets were sufficiently large for Parasol to find high quality (*i.e.,* near optimal) configurations. Training trace size mattered more for some applications than others. One large class of applications where training trace size mattered was applications that use hash tables. Here, traces had to be large enough to cause hash

collisions; otherwise the differences between configurations are small and it is difficult (or impossible) for Parasol's search algorithm to find the best one. For example, the Starflow configurations found by the simplex and Bayesian strategies resulted in similar eviction ratios (12% and 5%, respectively) in a small trace of 5000 packets, but had very different errors (46%, 26%) with a larger trace of 5 million packets.

It was often important that the training trace was representative of the testing trace. For some applications, the search phase was only effective when a trace contained certain network events. For example, the ConQuest data structure detects microbursts, and only begins monitoring when one occurs. A trace with no microbursts would produce no meaningful objective, regardless of the configuration.

Some applications, however, were less sensitive to differences between training traces and target workloads. When testing Starflow on a wide-area network (WAN) trace, we found that Parasol was able to find near-optimal solutions using training traces from either a WAN or a datacenter.

4.6.2.2 Comparison to hand-optimized configurations

We compared the performance of Parasol configurations to that of hand-tuned configurations for our three most complex applications: Fridge, ConQuest, and Starflow. The hand-tuned configurations come from the applications' original evaluations [18, 69, 86]. Our goal is to determine whether Parasol can essentially reproduce these results, by finding configurations that perform comparably on a similar workload.

Fridge (Unbiased RTT). The Fridge [86] data structure is used to collect RTT samples in the data plane by storing requests and matching them with the corresponding response, without sampling bias against large RTTs. Each request is added to the data structure with probability p, and once a request is in the structure, it can be removed either upon receipt of the response, or if a new response overwrites it due to a hash collision.

The value of p is the primary parameter to be optimized. If p is too small, requests are less likely to be added to the structure, and the program will not produce enough RTT samples. Conversely, if p is too large, requests are more likely to be overwritten before their responses arrive.

In general, the objective function that Fridge seeks to minimize is the difference between the distribution of sampled RTTs and the distribution of all RTTs. We implemented the same error function in Parasol as was used in the original evaluation of Fridge [86]: maximum percentile error, or the maximum error of the sampled distribution for percentiles $\in [5\%, 95\%]$.

In the hand-tuned program, the authors achieved an error of 25%, and our optimized program, found using Bayesian search, achieved a maximum delay estimation error of 28%. The Fridge authors found that they could achieve nearly the same error with a wide range of p values. In our workloads, Parasol also found that p had a negligible effect on error as long as it is greater than 2^{-12} (0.0002). Going outside of that bound for the chosen fridge size increased the error to over 100%.

ConQuest. ConQuest [18] aims to identify flows that are making a significant contribution to queue build-up, during some time window T. It maintains several sketches as "snapshots" of the queue length for T. During a time window, the program cleans one sketch, writes to one sketch, and reads a flow's queue length estimates from the rest.

ConQuest has three parameters that can impact its performance: the number of sketches and the rows and columns in each sketch. These parameters are challenging to tune because the choice of one affects the others. If the number of columns is too large, it reduces the number of rows that will fit on the target, and the sketch may not be fully cleaned before rotating. Conversely, too many rows requires less columns and smaller sketches. As a sketch gets smaller, it becomes less accurate.

The objective of ConQuest is to identify the packets responsible for queue build-up as accurately as possible. For comparison with the original evaluation, we quantify accuracy using the F-score[3], which depends on both precision and recall.

The original evaluation of ConQuest found that it could achieve both precision and recall greater than 90%, *i.e.,* an F-score >90%. Parasol found a comparable configuration with an F-score of 92% (precision of 97% and recall of 87%). The Parasol optimizer used the Bayesian search strategy.

The choice of metric used for cost affects the configuration chosen by the optimizer. F-score incorporates both precision and recall. A configuration with lower precision has more false positives, and a lower recall means more false negatives. Some applications may be more tolerant to false negatives, and others may prefer false positives. We can tailor the objective function based on an application's preference.

To minimize false positives, we can optimize for precision. This will result in a larger sketch, that keeps more accurate counts for each flow. On the other hand, we can optimize for recall to minimize false negatives. This produces a configuration with a smaller sketch, which will result in more flows being identified as significant contributors. In other words, more true positives, at the cost of more false positives as well.

Starflow. Starflow [69] is a telemetry system that partitions query processing between the data plane and software. The switch selects and groups per-packet records, which are sent to software for flow-level analytics (*e.g.,* classifying traffic, identifying microbursts). Packet records are stored within buffers on the switch, and are evicted to software when their buffer is filled, no buffer is available, or there is a collision. There are two kinds of buffers, whose sizes must be configured at compile time: a "narrow" buffer which tracks many small flows, and a "wide" buffer for tracking a few large flows.

[3]Specifically, the cost is 1 minus the F-score

App	Preprocess time	Train trace size, time	Test trace size, time
CMS	16s	500k, 25s	10M, 12min
MHT	15s	1M, 47s	10M, 7min
KV, Precision	25min	1M, 6min	5M, 25min
KV, CMS	2hrs	1M, 2min	5M, 7min
RTT	23s	1M, 1min	3M, 3min
Fridge	3s	1M, 1min	3M, 2min
Starflow	1.5hr	900k, 1min	5M, 27min
ConQuest	15s	10M, 9min	10M, 10min
LB	2s	500k, 16s	3M, 2min
Precision	32min	1M, 6min	18M, 1.7hrs
SFW	30s	4M, 3min	11M, 7min

Table 4.4: Runtime of Parasol components per application. Preprocess time is the total time to preprocess with the greedy layout heuristic, train/test trace size is the size of the trace in packets, and train/test trace time is the average time to simulate the trace once.

The most important performance metric for Starflow is its eviction ratio: the ratio of flushed cache records to packets. A lower eviction ratio is preferable because it means that more packets are being covered by each record that the server must process, saving both bandwidth and processing time at server.

The original, hand-optimized P4 code achieved an eviction ratio between 7.1% and 25%, depending on the size of the cache and the workload. The Parasol optimizer achieved an eviction ratio of 15%, well within the performance range of the original program. In other words, 15 out of every 100 packets are recirculated to evict a record from the cache. The best compiling configuration was found after 7 (out of 85) iterations (1.5 min) of simulated annealing. We found that both the sizes of the narrow and wide buffers impacted the eviction ratio. Our optimizer found, for our representative traffic trace, that a narrow cache smaller than 1024 slots and a wide cache smaller than 8192 slots resulted in an eviction ratio greater than 40%, with fixed wide and narrow caches, respectively.

4.6.2.3 Optimizer Speed

The runtime of the Parasol optimizer is application-dependent (shown in Table 4.4), and has two major components: preprocessing time and search time. Preprocessing time scales with the complexity of the input program and number of parameters, and took between 7 seconds and 1.5 hours. Search time scales primarily with the size of the input trace, and was limited to 2 hours, though many applications required less than that. A single iteration of the training trace took between 16 seconds to 9 minutes, depending on the application.

Overall, the Parasol optimizer took no more than 3.5 hours to find near-optimal settings for any of our applications. This compares favorably to compiling, testing, and tuning applications by hand: just compiling *one* configuration of a program to a reconfigurable architecture like the Tofino can take hours [28] for both research or industrial compilers, because it is a fundamentally hard task [76]. As mentioned above, we found three main factors that influenced the overall runtime: application complexity, training set size, and search strategy.

Application complexity. The optimizer preprocesses each Parasol program as a heuristic to check if it will compile to hardware. The preprocessing time depends on the complexity of the program, both in terms of length and number of parameters. Programs with more parameters (*e.g.,* Starflow) took longer than programs with few parameters (*e.g.,* LB). Table 4.4 lists total preprocessing time for each application.

Complex programs also take longer to simulate. The CMS simulation took about a minute for a 1 million packet trace, while a trace of the same size with Precision took three minutes. Precision is more complex because it contains logic for recirculating packets, while the CMS does not recirculate packets. The recirculation not only adds complexity to Precision, it also requires the program to process more packets, as recirculated packets must be processed again.

Training set size. The runtime of Parasol's search phase increases roughly linearly with the size of the input training trace, because the search algorithm executes each chosen configuration on the trace. Reducing the size of the provided trace can speed up optimization, but many applications require large traces. For example, evaluating the performance of a program that measures heavy hitters (*e.g.,* Precision) requires enough traffic that the trace contains heavy flows.

Search strategy. Search strategies took different amounts of time to converge, depending on the application. We compare search strategies, using the load balancing and Starflow applications, by tracking the best evaluated configuration after *each* iteration. All three methods found similarly performing configurations for the load balancer, but the overall search time was much different: Bayesian search took approximately 19 minutes, while simulated annealing and simplex search took only 2 minutes. Similarly, for the Starflow application Bayesian and simulated annealing strategies reached a configurations with similar performance (in 13 and 10 minutes, respectively) while simplex did not find a configuration that produced the best collision rate within the time budget.

4.6.3 Case Studies

4.6.3.1 Data-plane caching

To better understand how Parasol handles workload dependence and some of the challenges in tuning data-plane applications, we study a conceptually simple in-network cache. We optimize the cache for three different workloads: a highly skewed zipfian (top 10 keys had 58% of requests), moderately skewed zipfian (top 10 keys had 15% of requests), and uniform (top 10 keys had .06% of requests). Training traces contained 1 million requests, and test traces contained 5 million requests. We limit the cache size to 10K entries.

We compared three versions of the cache: a variant that uses a count-min sketch to track key popularities (NetCache [42]); one that uses Precision [5] to prioritize popular keys; and a very basic hash-addressed array that evicts on collision to avoid the need for packet recirculation.

Distribution	CMS	Precision	Hash Table
High skew	0.10	0.07	0.10
Moderate skew	0.69	0.64	0.70
Uniform	0.73	0.73	0.73

Table 4.5: Cache performance with respect to miss rates.

First, as Table 4.5 shows, all three caches reduce the workload of the backend that they serve. As expected, the caches perform better in more skewed workloads, and the more sophisticated CMS and Precision caches outperform the simple hash table. In particular, the Precision cache is over 30% more effective than the other caches, in the high skew workload.

Distribution	CMS	Precision	Hash Table
High skew	0.5750	0.5375	0.5500
Moderate skew	1.0175	0.8225	0.8500
Uniform	1.0475	0.8675	0.8650

Table 4.6: Cache performance with respect to network workload.

Now consider a network operator with a different objective. Instead of minimizing miss rate, they wish to minimize total network traffic. Assuming the client and server are connected via one hop across the caching switch, a cache miss costs 2X as much as a cache hit, and a recirculated packet costs 0.5X as much as a cache hit. Thus, the objective function is $2 * m + h + 0.5 * r$, where m, h, and r are the percentage of misses, hits, and recirculated packets in a trial. The cache provides benefit whenever the metric is less than 1.

Table 4.6 compares the caches with respect to this alternative metric. Somewhat surprisingly, the simple hash table performs *better* than the more sophisticated CMS

variant, and is competitive with Precision (even beating it for the uniform workload). It is because the hash table variant does not need to recirculate packets, unlike the others.

We also compare the results of optimizing the cache with both the P4All and Parasol frameworks. We use a weighted sum of the theoretical hit rate of the key-value store and the error rate of the CMS (described in 3.2: $0.8 * (kvObj) - 0.2 * (cmsObj)$. This objective function produces almost the same memory allocation that we get with Parasol. Memory allocation, however, is only a piece of the puzzle. The cache performance is impacted by the threshold we use to decide if an item should be inserted into the cache. P4All cannot optimize for the threshold, because it is a non-resource parameter, meaning its value does not affect the resource allocation. If we pick a suboptimal threshold, hit rate suffers. In the moderately skewed workload, the Parasol optimizer chose a threshold of 300 (*i.e.,* if the switch has seen at least 300 requests for an uncached item, that item should be added to the cache), which results in a miss rate of 66% with the testing trace. If we instead choose a larger threshold of 5000, miss rate increases to 68%. For the test trace of 5 million packets, that would result in an extra 100k packets being forwarded to a storage server, when they otherwise could have been handled by the switch if we set the threshold optimally.

This case study highlights how tricky it can be to tune even a conceptually simple data-plane application. The optimizations that at first seem most effective (or are most intuitive) are not necessarily best in every network, or from every perspective. Figuring out what's right for one's network can be challenging, but Parasol simplifies this process by lifting the burden of reasoning about how parameter choices can affect performance off of the programmer.

4.6.3.2 Fridge

Sometimes, operators tune their programs with heuristics derived from closed-form equations based on worst-case error bounds. Such heuristics have two problems. First, they are challenging to derive and verify, hence only available for certain data structures whose properties have been theoretically analyzed. Second, closed-form equations do not always give an accurate picture of application performance in practice, because actual traffic distributions can vary significantly from the worst case [16] and, for tractability, closed-form equations often ignore factors that matter in practice.

To illustrate this, we compare the performance of the Fridge RTT monitor as optimized by Parasol to a version optimized according to a closed-form equation. Given a Fridge size M (the number of entries), the authors derive the following formula to set p: $\frac{M}{p}$ = number of requests between the request and response with the maximum delay, where p is the probability of storing a new request in the structure.

For our evaluation workload, with a Fridge sized at $M = 2^{17}$ (the maximum size for our implementation on the Tofino), the theoretical formula calculated $p = 2^{-1}$, which resulted in a maximum percentile error of 31%. As expected, this was *not* the optimal configuration for this workload. Optimizing with Parasol improved the relative performance by 10%; Parasol recommended $p = 2^{-5}$, which resulted in an error of 28%.

In Service-Level Agreements (SLAs) with ISPs, delay requirements are often specified as a target distribution, or a maximum delay for a certain percentile. It is then essential for ISPs to be able to accurately measure the delay distributions in their networks. The theoretical formula provides only a worst-case error bound, though, and it is not tailored to the more specific needs of users. Parasol, on the other hand, can easily optimize for users' target SLAs; programmers need only adjust the objective function. As such, the gap between Parasol and the theoretical formula was even

more substantial at specific percentiles—for RTT samples in the 50^{th} percentile, the configuration from the theoretical formula produced an error of 15%, while Parasol achieved an error of only 8%.

Although closed-form equations based on worst-case analysis are important for theoretical rigor, this short case study demonstrates that using them for tuning leaves significant performance on the table. Parasol allows network operators to reclaim that potential performance by automatically tuning data structures for different operating environments and performance objectives, while at the same time freeing programmers from the burden of deriving tuning heuristics from worst-case performance bounds.

4.7 Related Work

Researchers have developed a number of tools for writing data-plane programs. Domino [65], Chipmunk [28], ClickINC [82], Lucid [68], Lyra [27], and O4 [2] provide new, high-level languages for expressing data-plane programs, each providing abstractions and a compiler targeting one or more architectures. These compilers include optimizations or synthesis techniques to ensure that programs compile. However, if a program cannot fit on a target, it will not compile. In the case of ClickINC, the compiler will attempt to place the program on a different device if it cannot be compiled on a switch. They also do not provide environment-specific optimizations, as compilers do not have access to traces.

There also exist tools for optimizing prewritten data-plane programs. P^2GO [80] uses a traffic trace to minimize the resources used by a P4 program by reducing dependencies that do not appear in practice, shrinking tables, and offloading parts of the program to a controller. Cetus [47] uses static analysis to eliminate dependencies between tables by duplicating variables and to merge tables. Although P^2GO and

Cetus fit programs into limited resources, they either do not provide environment optimizations or risk changing program semantics. Additionally, Pipeleon [81] seeks to optimize the performance of programs deployed on SmartNICs by analyzing runtime performance. In constrast, Parasol focuses on resource-constrained programmable devices that cannot be updated at runtime without recompilation.

A third type of tool optimizes by leveraging user domain knowledge. P5 [1] uses a high-level description of the network's policy to remove spurious dependencies and unused features. SketchGuide [88] allows users to declare flexibly-sized structures and optimize them with a user-provided objective function. By taking policy into account, these tools can provide more detailed optimizations than would otherwise be possible. Although they provide a detailed optimization, these tools ask a lot of their users; P5 requires a high-level policy description, and SketchGuide requires a closed-form objective function to determine how their memory allocations relate to program performance.

An area of work related to Parasol's optimizer is network simulation. Simulators are designed for many objectives, including high fidelity [58], interactive operation [44], automatic traffic generation [85], and scalable performance [79]. All of these tools complement Parasol, and future work will likely involve integrating these tools to improve Parasol.

4.8 Conclusion

The process of writing and deploying a data-plane application that works well is an arduous one, requiring the programmer to undergo a grueling process of compiling, testing, and tweaking to find the best configurations. Parasol is a new and flexible framework for writing *parameterized* data-plane programs, and synthesizing effective settings for those parameters. Parameters in Parasol can represent a wide variety of

high-level implementation decisions, and the Parasol optimizer can target a variety of high-level behavioral goals. The optimization process is orders of magnitude faster than modern iterative testing strategies, and incorporates a representative traffic trace to tailor its solution to a particular environment. We evaluated Parasol on a variety of applications, and found that its solutions were near optimal and performed comparably to hand-optimized configurations.

```
1  /* A register array for cache hash table */
2  global Array.t<32> keyValue = Array.create(1024);
3  /* A register array for each CMS row */
4  global Array.t<32> counter0 = Array.create(2048);
5  ...
6  global Array.t<32> counter3 = Array.create(2048);
7
8  /* Execute the following on each packet */
9  event request(int key) {
10   int cachedKey =   // Hash key and return
11   int cachedTime =  // what's stored at that
12   int cachedValue = // index in the hashtable
13
14   /* Check if key in cache and if entry expired */
15   bool found = (key == cachedKey);
16   int timeDiff = Sys.time() - cachedTime;
17   bool expired = timeDiff > timeout;
18
19   if (found) { generate response(cachedValue); }
20   else if (expired) { AddKeyToCache(key); }
21   else {
22     /* Compute hash indexes for each row */
23     int cms_idx0 = hash(...);
24     ...
25     int cms_idx3 = hash(...);
26
27     /* Retrieve and increment stored counter values in each row */
28     int count0 = GetAndIncrStoredValue(counter0, cms_idx0);
29     ...
30     int count3 = GetAndIncrStoredValue(counter3, cms_idx3);
31
32     /* Calculate min across stored counter values */
33     int min = 0xffffffff; // Initialize global min
34     if (count0 < min)
35       { min = count0; }
36     ...
37     if (count3 < min)
38       { min = count3; }
39
40     if (min > threshold)
41       { AddKeyToCache(key); }  } }
```

Figure 4.4: Data-plane cache in Lucid.

```
1  symbolic size rows;
2  symbolic int cols;
3  symbolic int cacheEntries
4  symbolic int timeoutThresh;
5  symbolic int replacementThresh;
6  symbolic bool useCms;
7
8  module CMS : {
9    type t = ...;
10   fun t create(size rows, int cols) {
11     Array.t<32>[rows] cms = [Array.create(cols) for i<rows];
12     ...   }
13   fun int getCount(int key) {
14     int min = 0xffffffff ; // Initialize global min
15     for (i < rows) {
16       int cms_idx = hash(...); // Compute hash index
17       /* Retrieve and increment stored counter value */
18       int count = GetAndIncrStoredValue(cms[i], cms_idx);
19       /* Calculate min across stored counter values */
20       if (count < min) { min = count; }
21     } return min; }
22   }
23   fun bool decideIfAdding(int key) {
24     return (getCount(key) > replacementThresh);   } }
25 module Precision : {
26   type t = ...;
27   fun t create(size rows, int cols) = {...};
28   fun int getCount(int key) {...}
29   fun bool decideIfAdding(int key) {...} }
30
31 module KeyTracker=CMS if useCms else Precision;
32 global KeyTracker.t tracker = KeyTracker.create(rows,cols);
33 global Array.t<32> cache = Array.create(cacheEntries);
34
35 extern logHits(bool found);
36
37 event request(int key) {
38   int cachedKey =   // Hash key and return
39   int cachedTime =  // what's stored at that
40   int cachedValue = // index in the hashtable
41
42   bool found = (key == cachedKey);
43   int timeDiff = Sys.time() - cachedTime;
44   bool expired = timeDiff > timeoutThresh;
45
46   logHits(found);
47   if (found) { generate response(cachedValue); }
48   else if (expired) { AddKeyToCache(key); }
49   else {
50     bool add = KeyTracker.decideIfAdding(key);
51     if (add) { AddKeyToCache(key); }  } }
```

Figure 4.5: A data-plane cache in Parasol. Parts of the code not containing novel elements have been truncated or omitted entirely.

Chapter 5

Conclusion

Modern networks need to be able to support the complex applications, like network management and distributed services, that we rely on every day. Because traditional networks were designed primarily for communication between a small number of hosts, they are not capable of keeping up with our ever-changing needs and demand for high-speed processing. Programmable network devices have emerged as a way to bring increased capability and flexibility into traditional networks. These switches can be programmed to run a vast array of applications, which are guaranteed to execute at line-rate.

Programming these devices, however, can be prohibitively difficult. They guarantee line-rate execution because they have strict hardware limitations, and programmers often invest significant effort just to get their applications to fit within the resource constraints. Getting a program to fit is not enough—it must also perform well. Ensuring high performance requires balancing resource allocation across multiple applications, giving each enough resources, but not so much that the program violates constraints. This results in a tedious trial-and-error development process, because data-plane programming languages (P4) are too low-level. Programmers must

understand how their code gets mapped to hardware, and manually rewrite code to adjust the resource allocation.

This dissertation takes steps towards alleviating these challenges. We focus on decoupling the language from the hardware resources, by raising the level of abstraction for the languages, and building systems to automatically manage resource allocation. In this chapter, we provide a summary of contributions, along with future directions and final remarks.

5.1 Summary of Contributions

This dissertation focuses on decoupling data-plane programming languages from data-plane hardware. We build practical abstractions that allow for generalized data structures, that do not need to be rewritten for new applications. We automatically adapt those structures for specific contexts, with minimal programmer effort.

P4All (chapter 3) introduces elastic data structures, whose implementation is separated from the underlying hardware resource constraints. Elastic structures are defined by symbolic values, which function as placeholders for the size of the structure, and we extend the P4 language with these symbolic values. To automatically adapt these structures to whatever resources are available on the target, we develop an optimization framework to automatically set symbolic values according to a programmer-defined performance objective. This framework finds the optimal layout (*i.e.,* concrete values for each symbolic that optimize performance) by generating and solving an integer-linear program, where the constraints correspond to hardware resources and variables represent symbolic values. We implement the P4All optimizer in Python, and demonstrate that it can generate, in a matter of minutes, valid code that compiles to the Intel Tofino switch. We create a library of commonly used structures in P4All and find that the optimizer produces solutions comparable to

hand-optimized P4 code. P4All simplifies the process of data-plane programming by bringing more modularity and reusability to P4 and raising the level of abstraction.

Parasol (chapter 4) extends the concept of elasticity to encompass any parameter of a data-plane application, not just structure size. Parasol programmers write applications with symbolically-defined parameters (*e.g.,* structure size, timer values, thresholds, etc.) in the Lucid programming language. Programmers also supply an empirical objective function, written in Python, that expresses application performance as measurements taken during simulation. Parasol then uses a simulation-based optimization framework, in which it selects a configuration, simulates the application with a programmer-supplied traffic trace (using the Lucid interpreter), and measures the resulting performance. The output of the optimization process is a configuration which yields optimal performance, while adhering to hardware constraints. We evaluate the Parasol language by implementing ten diverse applications, each with differing parameters and objective functions. We measure the effectiveness and efficiency of the optimizer by comparing its output to optimal configurations found through exhaustive search, and we find that the optimizer is able to find near-optimal configurations for each of our applications within a two-hour time budget. We also compare Parasol and P4All, and show that Parasol is able to express a much wider range of applications. Even when we have both Parasol and P4All objective functions for the same application (*e.g.,* the Fridge data structure), the P4All objectives are less effective because they do not fully utilize workload properties. Parasol brings flexibility to data-plane programming through applications with generalized parameters, that can be optimized according to any criteria.

5.2 Future Directions

We can extend the concepts presented throughout this dissertation into new directions, that further promote a simplified development process for network applications.

5.2.1 Self-Measuring Data Structures

The Parasol optimizer uses user-provided traffic traces to tailor applications to particular environments. While this works well for consistent workloads, the traffic in a real network may deviate from the expected distribution. A deviation in performance or distribution may indicate that a structure is not well adapted to its environment or worse, that there is an attack on the network. In initial work in this space, experimental results show that the performance of an application can vary significantly based on the workload distribution. In other words, applications are less *efficient* for unexpected workloads. As such, these applications would significantly benefit from identifying, in real-time, deviations in distributions.

Self-measuring data structures efficiently and accurately measure distributions using data collected from the structures, alongside their intended functionality (*e.g.,* firewalls, load balancing, etc). As an example, a self-measuring count-min sketch would use the information stored in the sketch to identify if the estimated counts are likely to produce a high error, indicating the sketch is not performing as expected. The application could include a mechanism to send an alert to a network operator, but ideally, data structures would be able to adapt to changing workloads, without manual intervention. If the self-measuring sketch was identifying heavy hitters, it could potentially automatically adjust its classification threshold. These structures would then be more robust to potential attacks, and would provide better performance in the face of changing workloads.

5.2.2 Programming SmartNICs

SmartNICs [52, 55] are programmable accelerators that have become increasingly popular for offloading CPU tasks in data centers. They provide the ability to customize the networking stack, leading to efficient and flexible packet processing. Like programmable switches, smartNICs have certain hardware restrictions to enable efficient processing. Unlike switches, however, smartNICs do not adopt the all-or-nothing model; they utilize a run-to-completion model. If a particular program cannot execute at line-rate, it can still compile to and run on the smartNIC. This creates an interesting tradeoff for programmers to navigate—if they want more complex processing, it comes at the cost of reduced throughput.

We can apply the same strategies of P4All and Parasol to smartNICs. We need a comprehensive programming languages that allows programmers to write applications that can execute at line-rate, as well as more complex code that requires slower processing. We can also help programmers navigate this tradeoff by automatically finding the right balance of processing and throughput, so we can extract the best possible performance. This involves careful scheduling of operations on smartNIC cores and intelligently managing memory accesses.

5.2.3 Simplified eBPF Programming

eBPF [24] is a framework that allows programmers to write applications that can run in the Linux kernel, without having to change the source code or load kernel modules. eBPF is particularly useful for networking applications because it provides hooks to the network stack in the kernel. This allows programmers to customize packet processing and forwarding within the kernel, which enables faster processing, because packets do not necessarily need to go user space. For example, with eBPF, programmers can implement packet filtering, load balancing, and flow monitoring, among other applications, in the kernel. While eBPF is a powerful tool that simplifies

the process of running applications in the kernel, programming with eBPF is far from intuitive. Programs can run at different layers of the kernel, and the functionality available varies among layers. For example, the lowest layer, eXpress Data Path (XDP), sits at the earliest point, in the NIC, before packets are parsed. XDP is by far the fastest layer, but as a result, it is also particularly restrictive. It can only process ingress traffic, and it does not allow programming concepts such as loops, global variables, and floating point numbers.

We can extend the ideas of P4All and Parasol to develop a framework for programming with eBPF that allows programmers to write high-level, parameterized applications, without necessarily worrying about the layers in which the code can execute. Alongside this abstraction would be an optimizer that can automatically decide where the code executes. This is particularly challenging because of the performance tradeoff between layers. If programmers require more capabilities, their code must run in higher layers, which comes with a performance degradation. For example, the tc (traffic control) layer is 5-10x slower than XDP. A framework must take into account computational requirements of an application and the user's performance expectations. Lastly, the framework would also require a compiler that translates parameterized code into something recognized by the eBPF compiler.

5.2.4 Programming a Network of Heterogeneous Devices

Today's networks are comprised of many different types of devices—ASICs [20, 39], FPGAs [52], smartNICs [55], etc. Most of the devices use domain-specific languages. If programmers want to deploy the same application on different hardware, or to split an application across devices, this requires rewriting the program for each type of device. This makes network programming tedious and complicated. We can extend the concepts of flexibility and parameterization from P4All and Parasol to a broader range of devices. Programmers should be able to write a single program that can be

compiled to any network device with sufficient capabilities. In the case where a device cannot support the entire program, an optimizer should automatically determine which pieces of the program the device can support, while deciding where to place the rest of the program such that the original functionality is preserved and the resulting application meets the user's performance goals. The ultimate goal is to take full advantage of the programmability in the network by making it actually easy to program, without having to master device-specific languages for every device in the network.

5.3 Final Remarks

This dissertation simplifies data-plane programming by freeing programmers from the burden of reasoning about low-level hardware resources of programmable switches. Programmable switches, however, are just one piece of the puzzle. As we approach the ending of Moore's Law, highly specialized, domain-specific devices are becoming more common. Networks will need to adopt to a plethora of heterogeneous devices to keep up with increasing user demands. Unfortunately, because these devices are so specialized for high-speed processing, they also come with a high barrier to entry. They are not designed with usability in mind, with each device using different programming languages and having radically different constraints. Development of network applications can then become siloed, because programmers often become experts in a single type of device. Network-wide applications must be deployed across many types of devices, so development is broken up among several different groups of programmers. It becomes extremely difficult to adopt these devices in real networks, and we cannot reap their benefits in practice. We need to rethink the approach for programming these devices, and design systems that make them *actually* programmable, so that networks can easily adapt our changing needs.

Bibliography

[1] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. P5: Policy-Driven Optimization of P4 Pipeline. In *ACM SOSR*, 2017.

[2] Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, and Laurent Vanbever. Reducing P4 Language's Voluminosity Using Higher-Level Constructs. In *EuroP4*, 2022.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *ACM POPL*, 2014.

[4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing Oblivious Measurement Analytics. In *IFIP Networking*, 2020.

[5] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*, 2018.

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic In-band Network Telemetry. In *ACM SIGCOMM*, 2020.

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM SIGCOMM IMC*, 2010.

[8] Blackfire Technology. IMPACT Cyber Trust. `https://www.impactcybertrust.org`. Accessed: May 2024.

[9] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM CCR*, 2014.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[12] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *IEEE INFOCOM*, 1999.

[13] CAIDA. CAIDA 2016 Chicago direction A traces. `https://www.caida.org/catalog/datasets/monitors/passive-equinix-chicago/`. Accessed: May 2024.

[14] Center for Infrastructure Assurance and Security. National Collegiate Cyber Defense Competition. `https://www.nationalccdc.org`. Accessed: May 2024.

[15] Ritu Chadha, Thomas Bowen, Cho-Yu J Chiang, Yitzchak M Gottlieb, Alex Poylisher, Angello Sapello, Constantin Serban, Shridatt Sugrim, Gary Walther, Lisa M Marvel, et al. Cybervan: A Cyber Security Virtual Assured Network Testbed. In *IEEE MILCOM*, 2016.

[16] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. Precise Error Estimation for Sketch-Based Flow Measurement. In *ACM SIGCOMM IMC*, 2021.

[17] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *ACM SIGCOMM*, 2020.

[18] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-Grained Queue Measurement in the Data Plane. In *ACM CoNEXT*, 2019.

[19] Xiaoqi Chen, Hyojoon Kim, Javed M. Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring TCP Round-Trip Time in the Data Plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure (SPIN)*, 2020.

[20] Cisco Catalyst 9300 Programmable Switches. `https://www.cisco.com/site/us/en/products/networking/switches/catalyst-9300-series-switches/index.html`. Accessed: May 2024.

[21] Saar Cohen and Yossi Matias. Spectral Bloom Filters. In *ACM SIGMOD*, 2003.

[22] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms*, 2005.

[23] Graham Cormode and S. Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SIAM SDM*, 2005.

[24] eBPF. https://ebpf.io/. Accessed: May 2024.

[25] EX9200 Programmable Switches. https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html. Accessed: May 2024.

[26] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN ICFP*, 2011.

[27] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, 2020.

[28] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*, 2020.

[29] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM*, 2018.

[30] Gurobi Optimizer. http://www.gurobi.com. Accessed: May 2024.

[31] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM SOSR*, 2018.

[32] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe Elephants: Seize the Global Heavy Hitters. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure (SPIN)*, 2020.

[33] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*, 2022.

[34] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic Switch Programming with P4All. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[35] Mary Hogan, Devon Loehr, John Sonchack, Shir Landau Feibish, Jennifer Rexford, and David Walker. Automated Optimization of Parameterized Data-Plane Programs with Parasol. *arXiv preprint arXiv:2402.11155v1*, 2024.

[36] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *USENIX NSDI*, 2019.

[37] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM*, 2017.

[38] Qun Huang, Patrick P. C. Lee, and Yungang Bao. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *ACM SIGCOMM*, 2018.

[39] Intel Tofino. `https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html`. Accessed: May 2024.

[40] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the Fast Lane: A Line-Rate Linear Road. In *ACM SOSR*, 2018.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*, 2018.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, 2017.

[43] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.

[44] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[45] Lark Parser. `https://github.com/lark-parser/lark`. Accessed: May 2024.

[46] K. Rustan M. Leino and Philipp Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *TACAS*, 2010.

[47] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling. In *USENIX NSDI*, 2022.

[48] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[49] Zaoxing Liu, Ran Ben Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *ACM SIGCOMM*, 2019.

[50] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[51] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 2008.

[52] Mellanox Innova 2 Flex Open Programmable SmartNIC. `https://network.nvidia.com/files/doc-2020/pb-innova-2-flex.pdf`. Accessed: May 2024.

[53] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM*, 2017.

[54] Hun Namkung, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *USENIX NSDI*, 2022.

[55] Nvidia Bluefield-3 DPU. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf`. Accessed: May 2024.

[56] Recep Ozdag. Intel Ethernet Switch FM6000 Series–Software Defined Networking. Technical report, Intel, 2012.

[57] P4 Language Consortium. $P4_{16}$ language specifications. `https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.pdf`. Accessed: May 2024.

[58] George F Riley and Thomas R Henderson. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*. Springer, 2010.

[59] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM TOCS*, 1984.

[60] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the Network Be the AI Accelerator? In *ACM SIGCOMM Workshop on In-Network Computing (NetCompute)*, 2018.

[61] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent NetCore: From policies to pipelines. In *ACM SIGPLAN ICFP*, 2014.

[62] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous In-Network Round-Trip Time Monitoring. In *ACM SIGCOMM*, 2022.

[63] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *USENIX NSDI*, 2017.

[64] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX NSDI*, 2018.

[65] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*, 2016.

[66] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SOSR*, 2017.

[67] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ACM ASPLOS*, 2006.

[68] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM*, 2021.

[69] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow. In *USENIX ATC*, 2018.

[70] Stateful Firewall in P4. `https://github.com/p4lang/tutorials/tree/master/exercises/firewall`. Accessed: May 2024.

[71] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *USENIX NSDI*, 2021.

[72] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating Database Queries with Switch Pruning. In *ACM SIGMOD*, 2020.

[73] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *ACM EuroSys*, 2019.

[74] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *ACM SIGPLAN PLDI*, 2014.

[75] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *USENIX NSDI*, 2017.

[76] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms. In *EuroP4*, 2020.

[77] Vladislav D Veksler, Norbou Buchler, Blaine E Hoffman, Daniel N Cassenti, Char Sample, and Shridat Sugrim. Simulations in Cyber-Security: A Review of Cognitive Modeling of Network Attackers, Defenders, and Users. *Frontiers in Psychology*, 2018.

[78] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *ACM SIGCOMM*, 2013.

[79] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed Emulation of Software-Defined Networks. In *IFIP NETWORKING*, 2014.

[80] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 Profile-Guided Optimizations. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[81] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. Unleashing SmartNIC Packet Processing Performance in P4. In *ACM SIGCOMM*, 2023.

[82] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, and Bin Liu. ClickINC: In-Network Computing as a Service in Heterogeneous Programmable Data-Center Networks. In *ACM SIGCOMM*, 2023.

[83] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *ACM SIGCOMM*, 2018.

[84] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *ACM SIGCOMM*, 2020.

[85] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *ACM SIGCOMM*, 2021.

[86] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. Unbiased Delay Measurement in the Data Plane. In *SIAM APoCS*, 2022.

[87] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow Event Telemetry on Programmable Data Plane. In *ACM SIGCOMM*, 2020.

[88] Zhengyan Zhou, Jingwen Lv, Lingfei Cheng, Xiang Chen, Tianzhu Zhang, Qun Huang, Jiayu Luo, Longlong Zhu, Dong Zhang, and Chunming Wu. SketchGuide: Reconfiguring Sketch-based Measurement on Programmable Switches. In *IEEE ICNP*, 2022.