Automated Optimization of Parameterized Data-Plane Programs

Paper #569, 12 pages body, 15 pages total

ABSTRACT

Programmable data planes allow for sophisticated applications that give operators the power to customize the functionality of their networks. Deploying these applications, however, often requires tedious and burdensome optimization of their layout and design, in which programmers must manually write, compile, and test an implementation, adjust the design, and repeat. In this paper we present Parasol, a framework that allows programmers to define general, parameterized network algorithms and automatically optimize their various parameters. The parameters of a Parasol program can represent a wide variety of implementation decisions, and may be optimized for arbitrary, high-level objectives defined by the programmer. Furthermore, optimization may be tailored to particular environments by providing a representative sample of traffic. We show how we implement the Parasol framework, which consists of a *sketching language* for writing parameterized programs, and a simulation-based *optimizer* for testing different parameter settings. We evaluate Parasol by implementing a suite of ten data-plane applications, and find that Parasol produces a solution with comparable performance to hand-optimized P4 code within a two-hour time budget.

1 INTRODUCTION

The advent of programmable data planes has provided network operators the ability to customize the low-level behavior of network switches, paving the way for sophisticated applications that run inside the network itself. These applications include distributed firewalls, load-balancing, sophisticated telemetry, mechanisms for distributed coordination, and application accelerators like in-network caches. Each of these data-plane programs leverages the specialized packet-processing hardware of the switch to run at line rate.

This power comes at a price: sophisticated data-plane programs are notoriously difficult to write. Thanks to the complexity of switch hardware, programs written in domain-specific languages like P4 [5] often fail to compile [13–15, 24, 35], as they cannot fit their data structures into the limited memory and computation constraints available on a chip.

Even when they do compile, there are many performance-critical choices to make when crafting these algorithms—thresholds, memory allocation, which data structures to use, etc. The first version of an algorithm is rarely the best one. Consequently, applications often require many revisions and sometimes weeks' worth of tweaking and testing to obtain a variant of the initial program that compiles, fits within hardware constraints, and performs well. The process is all the more frustrating as compiling to programmable switch hardware can be very expensive: ASIC compilers can take anywhere from minutes to hours to days [14]. Indeed, in one author's experience, the state-of-the-art P4 compiler took *more than five days* to produce a working binary.

To alleviate these problems, researchers have developed a variety of different program synthesis systems aimed at programmable switches [13–15, 22, 32, 35]. These systems lift the level of abstraction at which programmers write programs, which gives those programs enough "flex" to be fit efficiently onto switches by optimizing the use of key resources such as memory, ALUs, or pipeline stages.

As useful as they are, these systems only scratch the surface of what is possible in a more general program synthesis framework for programmable switches. Each of the above systems is limited in one or more of the following ways.

Limited objectives. Most systems to date focus on optimizing simple on-switch resources. Common optimization criteria include memory footprint, number of stages, or ALU usage. However, humans typically evaluate network applications on far more sophisticated criteria than just whether they happen to fit into the switch pipeline: accuracy of measurements, effectiveness relative to an idealized model, and bandwidth used are just a few other ways to evaluate data-plane algorithms. Indeed, even the most memory-intensive applications are typically developed to optimize some *other* criteria. For instance, NetCache [16]—an in-network cache for key-value stores—uses several data structures, including a count-min sketch (to identify popular keys) and a multi-stage hash table (to cache the values for popular keys). While optimizing memory layout of these data structures is important, the high-level objective is actually to maximize cache *hit rate.* No tool to date has the ability to specify objectives at such a high level of abstraction.

Limited "program flex." To give optimizers a chance to improve a program implementation, they must be free to change that implementation — the more freedom (i.e., the more "flex" in the program), the more opportunities an optimizer has to make improvements. Standard optimizing compilers have very little freedom in this regard; they must preserve the surface-level semantics of programs. A system like P^{2}GO [32] deviates from this requirement by cutting out program components that are unnecessary for processing a particular traffic trace, but this carries some risk if traffic not present in the trace shows up in the live network. Even if it does not, P^{2}GO has limited ability to make changes, because it provides only three operations: it can merge tables, remove dependencies, or move processing to the control plane.

A system like P4All [15] or SketchGuide [35] adds a little more flexibility by allowing data structures to be resized. Still, data-plane algorithms could be tweaked in so many other ways that affect their performance: the rate at which active probes are emitted in a telemetry application, the choice of data structures to use in an in-network cache, the threshold at which to declare a heavy hitter, or the criterion to use for failure detection, to name just a few. No tool to date allows users to write programs with so much flex, let alone automatically optimize them.

Limited environmental input. The performance of a data-plane application is a product of its environment. A program tailored to one workload may not perform well if used in a different setting. As a result, systems like Chipmunk [14] and P4All [15] are limited because they have no access to traffic traces. Even if it were possible to express a property such as "optimize cache hit rate" in a system like P4All (which it is not), it would not be possible to solve the optimization problem because hit rate depends on the distribution of requests in the network, which P4All does not consider. P^{2}GO and SketchGuide *do* provide access to such data, but they have neither the flex nor the range of objectives to exploit that information to its fullest potential.

Developing an appropriate framework is challenging because of the tradeoff between expressibility and complexity. Making a framework more expressible by allowing more program flex inherently makes the optimization more difficult, as the program can have any number of parameters that affect its performance, and it can be nearly impossible to develop objective functions to capture every parameter in a program. On the other hand, while limiting the flexibility simplifies the optimization, it also restricts the types of programs that can be expressed.

Enter Parasol. Parasol is a novel, more general framework for synthesizing data-plane programs that overcomes the limitations of earlier frameworks. Parasol consists of two parts: a sketching language and an optimizer. The sketching language is an extension of Lucid [24], a high-level, event-based data-plane programming language. Parasol programmers write *sketches* [23], which are normal programs with several "holes". These holes 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. This might include memory layout, decision thresholds, measurement intervals, or even a choice between data structures; in contrast, systems like P4All and SketchGuide are limited to only optimizing memory layout.

The optimizer uses an iterative search algorithm to automatically optimize the parameters according to a user-defined 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.

To summarize, 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 developing a number of data-plane programs with various parameters and objective functions. Our experiments found that the Parasol optimizer completed an 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. The solutions produced by the optimizer not only complied with the resource constraints of the hardware, but were comparable in performance to hand-optimized P4 code.
Ethics: This work does not raise any ethical issues.

2 AN ILLUSTRATIVE EXAMPLE

Before describing Parasol in detail, we provide a motivating example application that one might wish to deploy in a programmable network: a load-balancing cache, inspired by NetCache [16]. The structure of the cache is illustrated in Figure 1. The cache reduces load on storage servers by directly

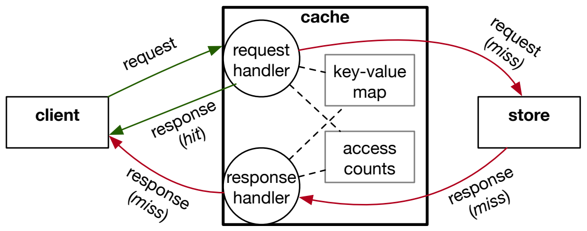

Figure 1: Motivating example: an in-network cache.

Param	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.

Figure 2: Parameters of the data-plane cache.

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, 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)).

Parameters and performance. The high-level description of the cache algorithm is quite simple, but to implement it, a programmer must make numerous low-level decisions. How much memory should be allocated to the hash table vs. the counter? When should we replace cached keys? How should we represent that counter—using a CMS, or something else? Consider a cache implemented using Precision [3], a hash table that probabilistically replaces cached keys upon collision, where more popular items are less likely to be replaced. Each of these questions corresponds to a *parameter* of the program; Figure 2 provides a non-exhaustive list of the parameters that an implementation might depend on.

These decisions are not simply details — they can have significant performance implications. For example, a larger hash table can cache more keys at once, but reduces the memory available to the approximate counter and, in turn, its accuracy. 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.

Fundamentally, these trade-offs exist because programmable switches have extremely limited resources that are shared across all data structures on the switch. As a result, it is particularly difficult to figure out precisely what effect different decisions will have on the program's behavior.

In contrast, the *desired* behavior of a data-plane cache is easy to define—it should maximize hit rate. This behavior is equally easy to measure, by simply monitoring the switch in question and recording whether each incoming packet is a hit or a miss. However, hit rate is not easy to *predict* from the values of the cache's parameters, let alone model analytically. It would be very difficult (likely impossible) to derive a closed-form equation that relates the cache's hit rates and parameters, which precludes us from using recent ILP-based optimization frameworks [15] to find a good configuration.

Traffic dependence. There is another wrinkle: the hit rate of the cache does not depend solely on the parameters, but also on the network. This can be illustrated by comparing Parasol's implementation of a cache against an equivalent implementation in P4All, another framework that optimizes parameters of a program sketch. The hit rate 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 parameter, neither of which can be optimized by P4All.

As a result, the performance of the P4All program is limited by the programmer's hard-coded choices for these parameters. 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, making it common to pick suboptimal values. We found that if the programmer chose those values poorly, the hit rate for a skewed workload could be as low as 56%, while our optimizer produced a solution with a hit rate of 93%. For a uniform workload that hit rate plunged to 11%, while Parasol managed a hit rate of 28%. One might worry that Parasol is achieving its better hit rates by overfitting to its input trace; this is a concern for any framework that relies on a particular input. We discuss how to prevent overfitting in detail in §4.4.

f

3 SKETCHING LANGUAGE

The first component of Parasol is a *sketching language* that allows users to write parameterized programs. This language is an extension of Lucid [24], a high-level data-plane programming language built atop P4. 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.

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.

To implement Parasol, we add three new features to Lucid. First, we add symbolic values (à la P4All [15]) 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 3 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, including the hash table storing the key/value pairs.

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.

The program in Figure 3 contains four symbolic values. The boolean `useCms` determines if the program should use a CMS or Precision data structure, and the integer `trackerSize` determines how much memory is allocated to that structure. If a CMS is used, `cmsThresh` determines the threshold for adding new keys to the cache. Finally, `timeout` determines when keys in the cache are considered expired.

Selecting Data Structures. Lucid provides a standard *module* system for representing data structures. Each module contains definitions for zero or more types, functions, and events. In Figure 3, 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.

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

```
symbolic bool useCms;
symbolic int trackerSize;
symbolic int cmsThresh;
symbolic int timeout;

module CMS : {
  type t = ...;
  fun t create(int size) {...}
  fun int getCount(int key) {...}
  fun bool decideIfAdding(int key) {
    return (getCount(key) > cmsThresh);   } }
module Precision : {
  type t = ...;
  fun t create(int size) = {}...};
  fun int getCount(int key) {...}
  fun bool decideIfAdding(int key) {...} }

module KeyTracker=CMS if useCms else Precision;

global KeyTracker.t tracker =
    KeyTracker.create(memSize);

extern logHits(bool found);

event request(int key) {
  int cachedKey =   // Hash key and return
  int cachedTime =  // what's stored at that
  int cachedValue = // index in the hashtable

  bool found = (key == cachedKey);
  int timeDiff = Sys.time() - cachedTime;
  bool expired = timeDiff > timeout;

  logHits(found);
  if (found)
    { generate response(cachedValue); }
  else if (expired)
    { AddKeyToCache(key); }
  else {
    bool add = KeyTracker.decideIfAdding(key);
    if (add) { AddKeyToCache(key); }  } }
```

Figure 3: A demonstrative implementation of a data-plane cache in Parasol. Parts of the code not containing novel elements have been truncated or omitted entirely.

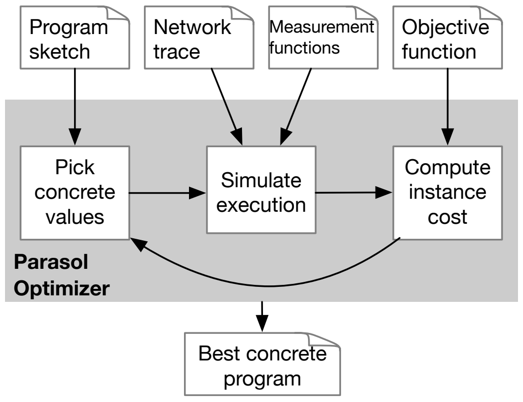

Figure 4: Overview of the Parasol optimization framework.

value `useCms`. 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.

Foreign Function Interface. Our final extension to Lucid 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 3, the extern `logHits` 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 effectiveness.

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.

4 OPTIMIZING SKETCHES

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. The programmer provides four inputs: (1) a program sketch, (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 sketch 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.

4.1 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.

4.2 Measurements and Objectives

Parasol optimizes each program according to a programmer-defined objective function, written in Python. 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.

Objective and measurement functions are often simple. For our data-plane cache, the goal is to minimize the miss rate (that is, the ratio of cache misses to cache accesses). The functions for measuring and computing miss rate can be defined in just eight lines of Python (Figure 5).

The measurement function `logHits` is called from the Parasol program once per request, as in Figure 3. The `objective` function is called by the optimization algorithm at the end of

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

```
hits = 0
misses = 0
def logHits(found):
  global hits, misses
  if found:   hits += 1
  else   :   misses +=1
def objective():
  return misses/(hits+misses)
```

Figure 5: Measurement and objective functions for the data-plane cache.

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.

Programmer effort. Optimizing a program based on cost and measurement functions greatly reduces programmer effort, compared to prior frameworks that optimize based on closed-form functions. Writing a closed-form function to represent the miss rate for a data-plane cache is an arduous task. 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 miss 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 miss rate also depends on the accuracy of the counts in the sketch.

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 [11]. *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 [8].

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 [10, 21]. 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 data structure is full, the switch cannot store new SYN packets; as a result, the 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 possibly perform.

4.3 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 configurations that do not compile, it also reduces the effectiveness of the search strategies because it does not give any indication of a direction in which to search. One could imagine simulating the configuration 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. 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 avoid compiling every solution in the useful 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 1. We choose a configuration from phase 1, 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 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. 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 optimization 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 strategies (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 the built-in strategies and analyze how the choice of strategy affects the optimizer in § 5.

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

Figure 6: The performance of each preprocessing heuristic for a single configuration, averaged over each evalauted application. The greedy layout provides the best balance between performance and accuracy.

4.4 Design Tradeoffs

Accelerating Preprocessing. The first phase of optimization requires us to analyze the resource usage of a program to determine if it will compile or not. The simplest way to do this would be to actually compile the program; however, compilation can be very slow (the Conquest [9] 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. We provide a detailed description of each heuristic in Appendix A.

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. They range in complexity from only considering dependencies between actions (dataflow graph), to modeling every resource except packet header vector (PHV) constraints (compiling from Lucid to P4).

A summary of the performance of our heuristics appears in Figure 6. 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 tradeoff 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 §5.4, 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 [27]. These traces come from a variety of sources, including cyber defense exercises [12] and security-oriented testbeds or simulators [7, 30].

5 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 the section, we discuss each Parasol component individually, and finish with an in-depth look at our data-plane caching example. We used three types of traces in our evaluation—the University of Wisconsin Data Center Measurement trace [4], a trace from core Internet routers [6], and synthetic traces for the cache application.

[2]https://anonymous.4open.science/r/Parasol-FED1/

5.1 Language

To evaluate the expressivity of Parasol, we implemented applications with multiple classes of parameters and diverse objectives. The right two columns of Figure 7 show the high-level benefit of Parasol over prior optimization frameworks: whereas Parasol allowed us to fully express the optimization goal of each application (parameters and objective function), P4All and SketchGuide 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 Figure 7 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 time between packets. Parasol's flexible approach allowed it to handle all of them.

In comparison, P4All and SketchGuide, the only prior frameworks for application-level parameter optimization, 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 or SketchGuide.

Even for the examples that *could* potentially be optimized by P4All or SketchGuide, 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 and SketchGuide.

Objective functions. The objective functions for our applications measured a wide variety of high-level properties (Figure 7). 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, existing systems (P4All, SketchGuide) require 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

	Classes of parameters in application					P4All/S.G.?	
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 [26])	√	√	√	√	Miss rate (23)	✗	✗
RTT monitor (RTT [10])	√			√	Read success rate (118)	✗	✗
Unbiased RTT (Fridge [34])	√	√			Max percentile error (88)	✗	√
Starflow [25]	√				Eviction ratio (17)	√	✗
Conquest [9]	√				F-score (101)	√	✗
load balancing (LB [28])	√	√			Error vs. optimal (38)	✗	✗
Precision [3]	√				Avg. error for top flows (28)	√	✗
Stateful Firewall (SFW [24])	√	√		√	Packet overhead (70)	✗	✗

Figure 7: Applications optimized with Parasol, showing which classes of parameters/objective functions were used, and which of them could be expressed in P4All or SketchGuide.

error bounds. This is common, even in research, where many data-plane applications are evaluated empirically, without finding provable theoretical guarantees [9, 25]. 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 or SketchGuide 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 or SketchGuide. 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 found that we were only able to express three out of our ten objective functions in P4All or SketchGuide. 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, and applications optimized for the worst case may not perform as well on practical workloads. 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 [8, 19, 35].

5.2 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 1.1% (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.

5.2.1 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.

5.2.2 The Parasol searcher. As this section describes, 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.

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 8), 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. The effectiveness of Parasol search phase depended on the training trace's size and representativeness.

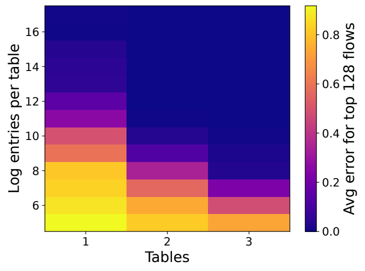

Figure 8: Average error for top 128 flows in the Precision application for different data structure configurations. A darker color represents a lower error. The optimal configuration achieved an error of 0.01%, and nearly 40% of the solution space produced an error of less than 1%.

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.

5.2.3 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 [9, 25, 34]. Our goal is to determine whether Parasol can essentially reproduce these results, by finding configurations that perform comparably on a similar workload.

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, 56s	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

Figure 9: 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 for an application.

Appendix B describes the case studies in detail, but at a high level, Parasol solutions performed reasonably close to the hand-optimized solutions for all three applications.

- For Fridge, Parasol found a configuration that achieved a delay estimation error of 18%, compared with the original evaluation's result of 25%.
- For Conquest, Parasol found a configuration with a precision of 97% and recall of 87%, compared to the original evaluation which found precision and recall > 90%, using the same trace.
- For Starflow, Parasol found a configuration with an eviction ratio of 15% in a wide area workload, which is better than the 18% eviction ratio reported in the original evaluation of the single configuration that the authors compiled to the Tofino.

5.3 Optimizer Speed

The runtime of the Parasol optimizer is application-dependent (shown in Figure 9), 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 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 [14] for both research or industrial compilers, because it is a fundamentally hard task [29].

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). Figure 9 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.

5.4 Case Study: Data-plane Caching

To better understand how Parasol handles workload dependence and how the distribution affects the performance of different structures, we provide a more detailed look at our cache application. We optimize the cache for three different workloads: skewed zipfian (top 10 keys had 58% of requests), less 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 first compare the cache with a CMS key tracker (NetCache [16]) and Precision key tracker. The skewness of the workload significantly impacted cache performance. For the skewed trace, the Precision cache slightly outperformed CMS (7% vs. 10% miss rate, respectively).

For the less skewed trace, Precision again slightly outperformed CMS (64% vs. 69% miss rate, respectively). However, they achieved the same miss rate for the uniform distribution (73% miss rate). The uniform distribution puts more pressure on the key tracker because there are more unique keys, causing worse performance than the skewed workload.

Both the Precision and CMS key trackers are complex applications that require recirculation to insert keys. As an alternative, we implemented a single-stage hash table as a cache, with no key tracker. The hash table always evicts keys on collisions, and thus does not require recirculation. For skewed and uniform workloads, the hash table performed similarly to Precision and CMS (10% skew, 73% uniform miss rate). Given that more complex structures provide marginal benefit, the hash version might be preferred for these workloads because of its simplicity. However, for the less skewed trace, the performance of the hash cache deviated more noticeably (70% miss rate). For this distribution, the added complexity of Precision provides a more notable improvement.

To mitigate overfitting, we also optimized our cache using a combination of the three traces. Our objective function was the average of the miss rates for each trace. The layouts chosen by Parasol for the Precision and hash versions were the same as when training with each workload individually, with Precision providing the lowest miss rate. The layouts chosen by Parasol for the CMS key tracker differed when trained on all three distributions. They performed slightly worse on each distribution individually — achieving miss rates that were approximately 1% worse for each distribution.

Parasol has the flexibility to express arbitrary programs with arbitrary parameters, that can be optimized with any objective, for any workload. With Parasol, we can directly compare different caching structures, for multiple traffic distributions, by simply tweaking a boolean value. In doing so, we found that the structure used in literature (CMS), is not always the best structure to track keys in a data-plane cache. Parasol is able to make this process easy because it lifts the burden of reasoning about how parameter choices can affect performance off of the programmer, greatly simplifying the development process for data-plane applications.

6 RELATED WORK

Researchers have developed a number of tools to more easily write data-plane programs. Domino [22], Chipmunk [14] Lucid [24], Lyra [13], 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, the program will not compile. They also do not provide environment-specific optimizations, as the compiler does not have access to traffic information.

There also exist tools for optimizing prewritten data-plane programs. P^2GO [32] 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 [18] uses static analysis to eliminate dependencies between tables 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.

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. P4All [15] and SketchGuide [35] allow users to declare flexibly-sized structures and optimize them with a user-provided objective function. Although they provide a detailed optimization, these tools ask a lot of their users; P5 requires a high-level policy description, and P4All and SketchGuide require a closed-form objective function.

An area of work related to the implementation of Parasol's optimizer is network simulation. Network simulators are designed for many objectives, including high fidelity [20], interactive operation [17], automatic traffic generation [33], and scalable performance [31]. In general, all of these tools complement Parasol. Future work on Parasol will likely involve integrating these tools to improve Parasol's simulator.

7 CONCLUSION

Currently, 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 settings for their program. 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 wide 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 networking environment. We evaluated Parasol on a variety of applications and objective functions, and found that its solutions were near optimal and performed comparably to hand-optimized configurations.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. 2017. P5: Policy-Driven Optimization of P4 Pipeline. In *ACM Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 136–142. https://doi.org/10.1145/3050220.3050235

[2] Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, and Laurent Vanbever. 2022. Reducing P4 Language's Voluminosity Using Higher-Level Constructs. In *International Workshop on P4 in Europe* (Rome, Italy) *(EuroP4 '22)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/3565475.3569078

[3] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE International Conference on Network Protocols*. 313–323.

[4] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *ACM SIGCOMM Internet Measurement Conference* (Melbourne, Australia) *(IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[6] CAIDA. 2019. CAIDA 2016 Chicago direction A traces. https://www.caida.org/catalog/datasets/monitors/passive-equinix-chicago/.

[7] 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. 2016. Cybervan: A cyber security virtual assured network testbed. In *MILCOM 2016-2016 IEEE Military Communications Conference*. IEEE, 1125–1130.

[8] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. 2021. Precise error estimation for sketch-based flow measurement. In *ACM SIGCOMM Internet Measurement Conference*. 113–121.

[9] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *ACM SIGCOMM Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3359989.3365408

[10] Xiaoqi Chen, Hyojoon Kim, Javed M. Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP Round-Trip Time in the Data Plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure* (Virtual Event, USA) *(SPIN '20)*. Association for Computing Machinery, New York, NY, USA, 35–41. https://doi.org/10.1145/3405669.3405823

[11] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[12] Center for Infrastructure Assurance and Security. 2023. National collegiate cyber defense competition. https://www.nationalccdc.org

[13] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. 435–450.

[14] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*. 44–61. https://doi.org/10.1145/3387514.3405852

[15] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Renton, WA, 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating System Principles*.

[17] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.

[18] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. 2022. Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Renton, WA, 371–385. https://www.usenix.org/conference/nsdi22/presentation/li-yifan

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

[20] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

[21] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *ACM SIGCOMM* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3544216.3544222

[22] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*. 15–28.

[23] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Architectural Support for Programming Languages and Operating Systems*. 404–415.

[24] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/3452296.3472903

[25] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow. In *USENIX Annual Technical Conference*. 823–835.

[26] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX Symposium on Networked Systems Design and Implementation*. 571–592.

[27] Blackfire Technology. 2023. Home. https://www.impactcybertrust.org

[28] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Boston, MA, 407–420. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

[29] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. 2020. Compiling packet programs to reconfigurable switches: Theory

and algorithms. In *P4 Workshop in Europe*. 28–35.

[30] Vladislav D Veksler, Norbou Buchler, Blaine E Hoffman, Daniel N Cassenti, Char Sample, and Shridat Sugrim. 2018. Simulations in cyber-security: A review of cognitive modeling of network attackers, defenders, and users. *Frontiers in Psychology* 9 (2018), 691.

[31] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. 2014. Maxinet: Distributed emulation of software-defined networks. In *IFIP Networking Conference*. IEEE, 1–9.

[32] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P2GO: P4 Profile-Guided Optimizations. In *ACM SIGCOMM Workshop on Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 146–152. https://doi.org/10.1145/3422604.3425941

[33] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: Fast performance estimates for data center networks with machine learning. In *ACM SIGCOMM*. 287–304.

[34] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. 2022. Unbiased delay measurement in the data plane. In *SIAM Symposium on Algorithmic Principles of Computer Systems*.

[35] Zhengyan Zhou, Jingwen Lv, Lingfei Cheng, Xiang Chen, Tianzhu Zhang, Qun Huang, Jiayu Luo, Longlong Zhu, Dong Zhang, and Chunming Wu. 2022. SketchGuide: Reconfiguring Sketch-based Measurement on Programmable Switches. In *IEEE International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940368

A PREPROCESSING HEURISTICS

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 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). 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 /emphany 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.

B COMPARISON TO HAND-OPTIMIZED CODE

We also compared Parasol solutions to hand-optimized solutions for three of our applications: Fridge, Conquest, and Starflow. Parasol's solutions performed reasonably close to the hand-optimized solutions for all 3 applications. We describe each application in more detail below.

Fridge (Unbiased RTT). The Fridge[34] data structure is used to collect RTT samples in the data plane by storing requests and matching them with the corresponding response, without penalizing samples with a large RTT. 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 when the structure is full.

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 [34]: 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 exhaustive search, achieved a maximum delay estimation error of 18%, well within the expected performance. 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 greater than 2^{-10} (0.001) or less than 2^{-3} (0.13). Going outside of that bound for the chosen fridge size increased the error to over 45%.

Conquest. Conquest [9] 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, and the configuration was found after 9 iterations.

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.

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

Starflow. Starflow [25] 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.

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.