NAP: Programming Data Planes with Approximate Data Structures

Mengying Pan
mengying@cs.princeton.edu
Princeton University

Hyojoon Kim
tcr5zr@virginia.edu
University of Virginia

Jennifer Rexford
jrex@princeton.edu
Princeton University

David Walker
dpw@cs.princeton.edu
Princeton University

ABSTRACT

Many applications that run on programmable data planes rely on approximate data structures, due to insufficient in-network memory. However, programming with approximate data structures is challenging because it requires (1) expertise in streaming algorithms to select the data structures that best match an application's requirements, (2) meticulous configuration to minimize approximation error while fitting within the hardware constraints, and (3) proficiency in the low-level P4 language. To address these issues, we propose NAP, a high-level network programming language. The core of NAP is the versatile *approximate dictionary* abstraction that captures a wide range of compact data structures, while allowing programmers to simply specify the kinds of error an application can tolerate. We demonstrate the language's expressiveness, conciseness, and efficiency through a variety of network applications, each compiling to P4 for the Intel Tofino in less than a second and featuring 25X-50X fewer lines of code compared to the P4 output. We evaluate an approximate stateful firewall written in NAP with real campus traffic, achieving performance consistent with the predicted accuracy.

CCS CONCEPTS

• **Theory of computation → Abstraction**; • **Networks → Programmable networks**; *Network resources allocation.*

KEYWORDS

Programming Language, Abstraction, Data Structure, Programmable Data Plane

ACM Reference Format:
Mengying Pan, Hyojoon Kim, Jennifer Rexford, and David Walker. 2023. NAP: Programming Data Planes with Approximate Data Structures. In *Proceedings of the 6th European P4 Workshop (EuroP4 '23), December 8, 2023, Paris, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3630047.3630196

EuroP4 '23, December 8, 2023, Paris, France
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0446-8/23/12.
https://doi.org/10.1145/3630047.3630196

1 INTRODUCTION

As networks continue to scale, there is a growing interest in controlling network traffic via programmable data planes. Programmable data planes enable a diverse set of high-speed network applications. These applications typically gather information from packet streams, and based on the information, make real-time decisions, which can involve dropping, reporting, modifying, or forwarding packets. Classic network applications include:

- A stateful firewall [18] that records outgoing flows to identify and drop unsolicited incoming packets;
- A layer-4 load balancer [15] that keeps track of millions of connections to servers to maintain connection affinity; and
- An in-network store [11] that detects and caches popular key-value pairs so that the queries of hot keys can be resolved in the data plane to reduce server workload.

Despite the advantages of programmable data planes, writing network applications for these targets is not easy. Many applications require taking different actions on different flows, but the data plane lacks sufficient memory to store information separately for potentially millions of flows. For example, a typical Intel Tofino [9, 10] switch has only tens of megabytes of register memory [12]. In contrast, a standard layer-4 load balancer requires hundreds of megabytes of memory [15].

As a result, network applications need to use approximate data structures to represent information compactly. For example, NetCache [11] uses a count-min sketch to estimate the number of queries for every key, occasionally overestimating less popular ones; for query resolution, it employs a finite-space cache, intending to serve cache hits for only the popular keys. Both data structures are space-efficient with approximation.

Despite the works that demonstrate the power of approximate data structures in the data plane [1, 13, 14, 16, 19], implementing and using them efficiently still imposes many challenges in practice. **Selecting the data structure.** With an expanding array of approximate data structures available, it is now a burden on users to choose the most suitable one. For example, while the original NetCache implementation uses a count-min sketch to track popular keys, it might instead have deployed a basic cache to keep counters for as many keys as possible and gather the popular ones from the keys in memory. Selecting the best data structure to use can be difficult, requiring a profound understanding of all the possibilities, especially when data-plane resources are limited.

NAP is available at: https://github.com/Princeton-Cabernet/NAP

Sizing the data structure. Once the decision is made, another challenge arises: optimizing the chosen approximate data structure to minimize the approximation error within hardware resource limitations. Manual execution of this process demands substantial time, effort, expertise, and iterative experimentation. Conventional compilers for programmable data planes, such as the P4 compiler for Intel Tofino, are not designed to search the implementation space efficiently.

Tailoring the data structure. Approximate data structures need to be contorted to satisfy the architectural constraints of the target. For example, a classical Bloom filter uses a register array, where multiple slots are accessed during insertion and querying. However, Intel Tofino restricts access to only a single memory slot. Hence, the Bloom filter needs to be tweaked to accommodate this constraint. Likewise, data structures often use sliding time windows to discard outdated information, but the PISA pipeline architecture [3] precludes a precise implementation. Without expertise in the specific hardware, writing an approximate data structure from scratch in a low-level language like P4 is error-prone and hard to debug.

To this end, we present **N**etwork **A**pproximate Data Structure **P**rogramming Language, or NAP, which is a high-level language for network programming with approximate data structures. The key novelty of the language lies in the abstraction for various kinds of approximate data structures, namely the *approximate dictionary*. An approximate dictionary stores key-value pairs. Typically, in the context of network applications, the key identifies a flow and the value represents the corresponding flow's information. The language provides approximations in two different dimensions:

(1) The *inclusion* dimension: A dictionary can either *overapproximate* or *underapproximate* the true data. Overapproximation occurs when unintended keys are included during a key insertion, expanding the dictionary's content beyond its space. Conversely, underapproximation happens when only a subset of key-value pairs can be inserted, adhering to the bounded size of the dictionary. Both of them are useful in various contexts.
(2) The *temporal* dimension: A dictionary is subject to either a *tumbling window* or an *approximate sliding window* to confine the collected information to the recent past.

With these abstractions, users can program approximate data structures solely by specifying and utilizing approximate dictionaries in NAP, without worrying about choosing a data structure, configuring its sizes, or writing its low-level implementation.

The NAP compiler chooses and configures approximate data structures in translating programs to P4. After a data structure choice is made, it finds the configuration with minimal approximation error while fitting into the hardware resource constraints. By specializing in data structure synthesis and thus narrowing the search space in advance, the compiler expedites this constrained optimization simply with a brute-force search and a greedy placement algorithm. Specifically, given a NAP program, the compiler first selects the underlying data structure for every approximate dictionary and computes the analytical error for all the possible configurations. Starting from the configuration with the lowest error, it then simulates a greedy placement of the program onto the Intel Tofino. The first configuration that fits is deemed the best and used to generate the P4 program.

We prototype NAP by extending the syntax of Lucid [18], which is a high-level network programming language on top of P4. We develop three common approximate dictionary classes—ExistDict, CountDict, and FoldDict—and apply them to write a diverse selection of network applications. We show that all the programs are compiled to P4 within a second and have 25X-50X fewer lines of code than their P4 outputs. Finally, we conduct a case study by loading the P4 program generated from an approximate stateful firewall written in NAP on the Intel Tofino and replaying a campus network trace. Our results show that its performance matches the analytical error predicted by the compiler.

Ethics statement: This study uses anonymized campus traces. Human network operators inspected all packet traces to ensure personal data were removed before being access. It has been conducted with all necessary approvals from the institute.

2 THE NAP LANGUAGE

Network applications often group traffic into flows to collect information. This makes the *dictionary* a natural abstraction, where a flow is represented by a key and the associated data by a value. To reflect the inherent approximation, we introduce the *approximate dictionary*, where the nature of approximation is denoted by the *error direction* (e.g., under- or over- approximation). Many network applications need a specific error direction when using approximate data structures. In this section, we demonstrate how approximate dictionaries facilitate programming with approximate data structures, using concrete examples in the NAP language.

2.1 Example: approximate stateful firewall

Enterprise networks often use a stateful firewall to drop unsolicited traffic from the Internet. In our example, it uses the pair of internal and external IP addresses as the key to identify a flow. In a stateful firewall, outgoing packets are always permitted, while incoming packets must match the keys of recent outgoing packets, say from the last 60 seconds, or they are deemed *unsolicited*. By recording the keys of outgoing packets, the stateful firewall scans the incoming traffic to drop any packet that does not have a matched key.

However, due to the limited hardware memory and the high traffic rate, approximation becomes necessary. One important observation is that many applications have a preference on the direction of the error. An enterprise network can tolerate the occasional acceptance of unsolicited packets over the risk of blocking legitimate traffic. This preference arises because other devices, such as an intrusion prevention system or the end hosts, can drop the small fraction of unsolicited traffic that bypasses the firewall. In contrast, blocking legitimate traffic, would wrongly disrupt normal connections. The approximate stateful firewall serves as a filter that drops the majority of unwanted traffic.

2.2 Approximate dictionary

The core abstraction of NAP is the approximate dictionary, mapping a key to a value. The key comprises header fields and other data to identify a flow, and the value holds one or multiple numerical fields to store flow information. A dictionary can be `created`, `added` a packet by the key, `queried` by the key, or simultaneously added and queried (`add_query`).

When adding a packet to a dictionary, if its key maps to no value, the value is initialized; otherwise, the existing value is updated. When querying a dictionary with a key, if it maps to a value, the value is converted into a single numeric result and returned; otherwise, nothing is done.

Our NAP prototype supports three dictionary classes.

- `ExistDict` stores keys and is queried for key existence.
- `CountDict` counts the number of packets for every key.
- `FoldDict` initializes, updates & reads values in a customized way.

The example below shows the first half of an approximate stateful firewall implemented in NAP. We define an `ExistDict` dictionary, keyed by the internal and external IP pair (line 1-5). We use this dictionary class to store the IP pairs of recent outgoing packets.

```
1 type key = {int<32> int_ip; int<32> ext_ip}
2 global ExistDict.t<key> seen =
3   ExistDict.create(over,
4                    within(sec(60), sec(90)),
5                    Exist());
```

All the classes of approximate dictionaries are created with three parameters. With the third one discussed later in Section 2.3, we are introducing the two main parameters here.

Error direction. The first argument to `create()` is the error direction. A dictionary stores key-value pairs. Ideally, programmers expect an exact 1-to-1 mapping between a key and a value (Figure 1a). There are in general two ways to approximate: *overapproximation* or *underapproximation*. An overapproximate dictionary maps multiple keys to a single value so that the data of these packets is accumulated in one value (Figure 1b). This error direction guarantees that there is always a query result at the cost that it is an "overestimation" of the true value. In contrast, in an underapproximate dictionary, a key is optionally added: it may be added or it may not (Figure 1c). In other words, it ensures that the query result, if it exists, is always exact, at the cost of providing no information for the rest of the keys. It is also possible to allow for both directions of errors, leading to a general *approximation* (Figure 1d). For the stateful firewall example, we allow mistakes of permitting unsolicited packets, so there can be more keys recognized as existing in the ExistDict than inserted. Hence, the error direction is overapproximation (line 3).

Time window. The second argument to `create()` is the time window. As the value of stateful information diminishes over time, programmers naturally want the old key-value pairs to expire from the dictionary. In stream processing, two primary windowing constructs prevail: a tumbling window (where time is split into contiguous disjoint intervals) and a sliding window (which is a fixed-length time interval ending at the current time point). Due to the limited computational resources, we introduce an *approximate sliding window* of a variable length within a user-defined range. Assuming that the current time is *curr* and the approximate dictionary is D, NAP supports the following time windows (Figure 2).

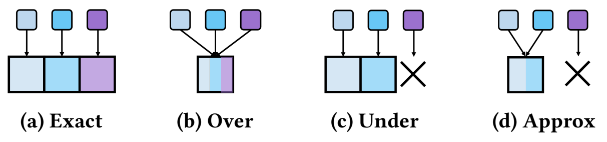

Figure 1: Error directions.

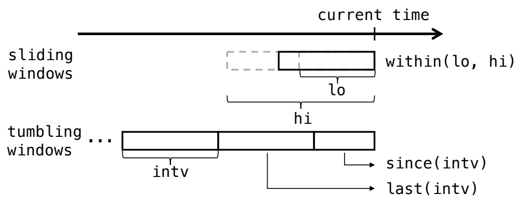

Figure 2: Time windows.

- `within(lo,hi)`: a sliding window of any duration $\in [lo, hi]$, so that $\forall p \in D, 0 \leq curr - p.time < t, lo \leq t \leq hi$.
- `since(intv)`: the latest tumbling window of duration *intv*, so that $\forall p \in D, 0 \leq curr - p.time < t, t = curr \mod intv$.
- `last(intv)`: the latest complete tumbling window of duration *intv*, so that $\forall p \in D, t \leq curr - p.time < t + intv, t = curr \mod intv$.

The stateful firewall adopts a sliding window between 60 and 90 seconds (line 4). It guarantees that all outgoing flows from now to 60 seconds ago are included in the dictionary and those older than 90 seconds are never included.

In the second half of the approximate stateful firewall program, we use the ExistDict to drop unsolicited packets. Any outgoing packet is added to the ExistDict (line 10-12), and any incoming packet queries the dictionary for the existence of its internal and external IP pair during the last 60-90 seconds (line 14-16). If the query returns false, the packet is deemed unsolicited and dropped (line 19). Since the dictionary is overapproximate, the query may have false positives, hence falsely allowing some unsolicited packets.

```
6  handle pkt_in(pkt_ty p) {
7    bool s = true;
8    if (p.ingress_port == INT_PORT)
9    then {
10     ExistDict.add(seen,
11                   {ext_ip = p.ip.dst;
12                    int_ip = p.ip.src}); }
13   else {
14     s = ExistDict.query(seen,
15                         {ext_ip = p.ip.src;
16                          int_ip = p.ip.dst}); }
17   if (s)
18   then { p.drop_ctl = NO_DROP; }
19   else { p.drop_ctl = DROP; }
20 }
```

2.3 Value state machine

When a key is added to a dictionary, the value is initialized if the key maps to no value and updated otherwise. In general, this can be described as a state machine, which involves

- an initialization function that sets up the state with packet data;
- an update function that computes a new state value from the old state and packet data;
- a reading function that converts the state to a numeric result.

The third argument of `create()` is such a state machine for values, which is commonly defined with `Fold` in functional programming languages [17]. For example, the state machine for an ExistDict can be expressed as `Fold(init, upd, read)` below. Since all ExistDict instances share this state machine, we can abbreviate it with `Exist()` (line 5).

```
type state_ty = {bool b}
fun state_ty init(pkt_ty p)
  { return {b = true}; }
fun state_ty upd(pkt_ty p, state_ty s)
  { return s; }
fun bool read(state_ty s)
  { return s.b; }
```

Predefined state machines cannot satisfy all the programming intents. Therefore, to allow users to customize their own value state machines in an approximate dictionary, we provide `FoldDict`. We show as an example below a FoldDict that counts the number of out-of-order (OOO) packets in TCP flows. The state here is a record of two integer fields, where the `fst` field stores the last seen TCP sequence number, and the `snd` field stores the number of OOO packets (line 3). After specifying the `init` (line 4-5), `upd` (line 6-9), and `read` functions (line 10-11), a state machine is defined, which will return the number of OOO packets as the query result (line 12-13). This FoldDict can be used to alarm possible network congestion when a TCP flow has too many OOO packets.

```
1  type key_ty = {int<32> src_ip; int<32> dst_ip;
2                 int<16> src_port; int<16> dst_port}
3  type state_ty = {int<32> fst; int<32> snd}
4  fun state_ty init(pkt_ty p)
5    { return {fst = p.tcp.seq_no; snd = 0}; }
6  fun state_ty upd(pkt_ty p, state_ty s)
7    { return {fst = p.tcp.seq_no;
8              snd = s.snd + 1 if s.fst > p.tcp.seq_no
9                              else s.snd}; }
10 fun int<32> read(state_ty s)
11   { return s.snd; }
12 global FoldDict.t<key_ty> ooo = FoldDict.create
13   (under, last(sec(60)), Fold(init, upd, read));
```

3 COMPILING TO THE DATA PLANE

We implemented NAP targeting the Intel Tofino. Following the PISA architecture, the Tofino data plane has several limitations. First, it is organized as a fixed number of stages, each allowing a bounded amount of simple computation. Also, every stage contains a small number of local register arrays, each with constrained memory capacity. Finally, a stage can only access its own register arrays, and such access is restricted to a single slot within each array.

An approximate dictionary implementation must adhere to these architectural constraints. To make full use of the memory, the data structure should feature multiple *rows*, each implemented on a register array (Section 3.1).

Furthermore, we must accurately remove outdated key-value pairs that exceed the time window. However, the constraint of a single access to each register array hinders bulk cleaning. Instead, we employ a technique from ConQuest [5], which involves creating multiple copies of a data structure and using packets to passively clean one entry at a time[1]. Each copy thus represents a *pane* in the time window (Section 3.2).

When implementing a multi-pane multi-row approximate data structure, we encounter several elastic parameters: the number of panes (P), rows per pane (R), and slots per row (S). Our NAP compiler optimizes these parameters to reduce approximation error while fitting the data structure within the data plane (Section 3.3).

[1]If the packet arrival rate does not support full pane cleaning in time, control packets can be generated and recirculated solely for cleaning purposes.

3.1 Selecting the data structure

Given a created approximate dictionary, the NAP compiler chooses the suitable data structure based on the dictionary class—ExistDict, CountDict, and FoldDict—and the error direction—exact, over, under, approx (Table 3).

An exact dictionary is implemented with an *exact cache* indexed by the key. Due to the 1-to-1 mapping, it requires $O(2^{|key|})$ memory, where $|key|$ is the length of the key. If its memory requirement exceeds the capacity, the compiler fails and suggests programmers add approximation.

An overapproximate dictionary ensures that every key has a value in the memory by hashing the key to a shared slot in a register array. This class of dictionaries can be implemented by either a *hashing cache* or a *sketch*. In a hashing cache, a given key is hashed to a single slot in a single row. It uses two hash functions to determine the row index and the slot index individually for accessing the corresponding state machine based on the key. In comparison, in a sketch, a key is hashed to a slot in every one of the R rows, each with its unique hash function. This type of data structure applies to the special case where values can be aggregated to reduce the error introduced by hash collisions and thus achieve higher accuracy. For instance, the compiler chooses a multi-row Bloom filter [4] for an overapproximate ExistDict. When inserting a key, each row sets to 'true' the hashed slot; when queried with a key, R values are read from all the rows and AND-ed to yield the final result. Since other keys may be hashed and inserted into the same slots, a key can be falsely identified as existing even if not added, introducing a false positive error. Similarly, a count-min sketch [6] is available for CountDict with an overestimation error.

An underapproximate dictionary guarantees that a value is accessible to a single key. This requires a *cache with full fingerprints*, where the entire key serves as a fingerprint stored alongside the value. Despite potential hash collisions when generating a hashed index, the fingerprint preserves the original key for identification, ensuring that querying retrieves a result only when the key matches the fingerprint.

A generally approximate dictionary can use any of the aforementioned data structures. Besides, for a cache with fingerprints, the full fingerprint is no longer necessary. Any partial fingerprints generated by hashing can be used since both error directions are allowed. Since there is no easy way to compare data structures with different error directions, the NAP compiler defaults to a Bloom filter for ExistDict, a count-min sketch for CountDict, and a *cache with partial fingerprints* for FoldFict.

Error dir.	ExistDict	CountDict	FoldDict
Exact	exact cache		
Over	Bloom filter	count-min sketch	hashing cache
Under	cache w. full fingerprints		
Approx	**Bloom filter**, all of the rest above, cache w. partial fingerprints	**count-min sketch**, all of the rest above, cache w. partial fingerprints	All of above, **cache w. partial fingerprints**

Table 3: Data structure choices.

3.2 Time window implementation

NAP uses a time window to define the eligible key-value pairs. To realize this abstraction, each data structure comprises a series of P panes, cycled through for reading, writing and cleaning operations. This technique enables both sliding and tumbling windows.

Approximate sliding windows. An approximate dictionary of `within(lo, hi)` captures key-value pairs from the present to lo in the past while excluding those older than hi in the past. To implement this, time is discretized into P segments, where $P \geq \lceil \frac{hi}{hi-lo} \rceil + 1$. Each pane stores information for a distinct time segment. Figure 4 illustrates the approximate sliding window scheme for a Bloom filter. At any given time, one pane undergoes cleaning and one undergoes writing. If a key is queried, $(P-1)$ values are read from the panes and OR-ed to find if the given key exists in any of the corresponding time segments. As one pane is written and read simultaneously, the approximate sliding window extends until the panes rotate and the oldest one expires.

Tumbling windows. The `since(intv)` time window is a base case of `within(lo,hi)` where `lo` is zero. This scheme can be realized with two panes by always reading and writing to the same pane. The `last(intv)` time window needs to keep information for both the current and the previous intervals, requiring three panes of a data structure as a result.

3.3 Sizing the data structure

An approximate data structure inherently introduces errors, with its configuration directly determining the error size and resource requirements. This configuration is characterized by parameters such as the number of panes (P), rows per pane (R), and slots per row (S). All of these parameters can be flexibly adjusted to minimize the error while fitting onto the data plane.

Due to the limited hardware resources, the configuration parameters can only be chosen from a finite range of values. For example, the number of rows R and panes P should satisfy that $R \times P$ is less than the total number of register arrays. Additionally, the number of slots S has to be a power of 2, as generating the hashed index otherwise requires an expensive modulo operation. Consequently, all the feasible configurations of (S, R, P) can be exhaustively enumerated.

For each configuration, the NAP compiler then calculates its analytical error. Table 5 provides the error functions for all the data structures in our prototype, which depend on only a few configuration parameters and input traffic characteristics. Besides S, R and P, some may rely on the fingerprint length (F) and the average number of distinct keys in the given time window (M). In cases where multiple approximate dictionaries are created in an application, the compiler combines all the errors into an overall objective.

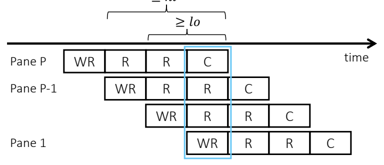

Figure 4: An approximate sliding window Bloom filter with *C*leaning, *W*riting and *R*eading panes.

Data structures	**What to minimize**	**Analytical error**
Bloom filter	False positive rate	min. $1-(1-(1-(1-\frac{1}{S})^{\frac{M}{P}})^R)^P$
Count-min sketch	Upper bound of expected error	min. $e^{-R}+\frac{e}{S}$
Cache w. full fingerprint	Cache misses	min. $M-SR$
Cache w. partial fingerprint	Key collision probability	$f=S(1-(1-\frac{1}{S})^{\frac{M}{R}})$ $k=\frac{M}{fR}$ $c=(1-\frac{1}{2^F})^{k-1}$ min. $Rf(k-c)/M$

Table 5: Analytical errors of data structures.

Following the computation of analytical errors, the configurations are ranked in ascending order based on error size. The NAP compiler then simulates the available hardware resources and attempts to place each configuration greedily onto the target until it identifies the first suitable fit.

Finally, the approximate data structure choices and size parameters, along with the rest of the NAP program, are translated into a P4 program. Our prototype predefines some commonly used packet parsers, assuming that users will provide additional definitions if needed. We leave the simplification of this task as future work.

4 EVALUATION

We evaluate NAP by implementing a variety of applications in Table 6 and compiling them to P4. We analyze the expressiveness of the language and the efficiency of the compiler. Finally, we run the stateful firewall on an Intel Tofino switch and evaluate its performance on a campus traffic trace.

4.1 Language design

We implement a diverse set of nine example applications in NAP, including network telemetry, network monitoring, and network control applications. Each of them utilizes one or more approximate dictionaries. All of these practical applications can be expressed within 30 lines of code (LoC). In comparison to their compiled P4 counterparts, NAP substantially reduces programming effort, achieving a reduction of 25X to 50X in LoC (Table 6).

It is worth noting that NAP generates highly modularized P4 programs, potentially resulting in an even greater reduction in LoC when compared to hand-written P4 programs. For reference, Lucid [18] generates a 2267-LoC P4 program for an approximate stateful firewall, significantly longer than our 555-LoC P4 output.

4.2 Compiler performance

All example programs compile to P4 programs for the real hardware target within one second, with single-dictionary applications taking less than 0.01 seconds to compile. To further assess our compiler's performance, we conduct benchmarks with an increasing number of overapproximate ExistDicts, which are compiled into Bloom filters. The compilation time roughly increases by a factor of 100 with each additional Bloom filter in the application (Table 6).

A closer look at the time breakdown reveals that, as the number of data structures grows, the majority of time is spent creating and fitting configurations. For instance, compiling the 4-Bloom filter benchmark takes on average 2278.87 seconds, with 2020.27 seconds dedicated to generating 261 million configurations and 255.23

Application	LoC		Compile time (s)	Configs	
	NAP	P4		Total	Opt. rank
Single dictionary					
Stateful firewall [2]	15	555	0.0055	525	87 (16.6%)
DNS amplification mitigation [2]	15	582	0.0056	525	87 (16.6%)
FTP monitoring [2]	20	798	0.0035	64	32 (50.0%)
Heavy hitter detection [2]	8	595	0.0049	126	3 (2.4%)
Traffic rates measurement by IP/8	12	466	0.0040	14	1 (0.8%)
TCP out-of-order monitoring [17]	19	559	0.0043	66	22 (33.3%)
Heterogeneous dictionaries					
TCP superspreader detection [2]	20	842	0.0130	3274	730 (22.3%)
TCP SYN flood detection [2]	20	842	0.0130	3274	730 (22.3%)
NetCache [11]	22	802	0.0394	9726	5049 (51.9%)
Bloom filter benchmarks					
1-Bloom filter	17	555	0.0055	525	87 (16.6%)
2-Bloom filter	30	960	0.1743	88053	4053 (4.6%)
3-Bloom filter	43	1289	25.94	6648.2K	367.4K (5.5%)
4-Bloom filter	56	1618	2278.87	261.0M	24.5M (9.4%)

Table 6: Network applications and benchmarks.

seconds spent on attempting to fit the top 9.4% of the configurations. This also highlights potential avenues for improving compiler performance in the future, either by reducing the total number of configurations or by expediting the discovery of the optimal one.

4.3 Stateful firewall case study

Our approximate stateful firewall example is compiled to P4, utilizing a 4-pane and 3-row Bloom filter with an analytical error of 0.329%. After deploying the P4 program on a Tofino switch, we replay a 10-minute anonymized trace captured at the campus border starting at 2 p.m. EST on August 19, 2020. The trace comprises around 106 million packets at a rate of 185,000 packets per second.

Compared to the ground truth with an exact 60-second sliding window, our approximate stateful firewall exhibits a 0.509% false positive rate. This error is slightly higher than the analytical error, possibly due to the temporal approximation. The NAP program adopts a $[60, 90]$-second approximate sliding window, which includes keys that the ground truth does not have in its exact window.

Figure 8 presents the variation of the false positive rate over time. It starts off at 0% since the initially empty Bloom filter does not allow any unsolicited incoming traffic. As it fluctuates between 0.25% and 2%, an interesting pattern emerges, characterized by spikes occurring about every 30 seconds, corresponding to the 30-second pane length. The rate gradually increases as the writing pane fills up and then drops suddenly when the oldest pane is replaced with a cleaned pane.

Table 7 displays the resource utilization for the output P4 program on the Tofino. While one might intuitively expect the program to fully utilize the memory, as well as the associated hash and ALU units, to minimize errors, the optimal allocation actually consumes only 25.6% of the SRAM. This deviation can be attributed to two primary factors: first, a significant portion of Tofino's SRAM cannot be allocated to register memory, and second, the preprocessing and postprocessing dependencies confine the data structures to

Resource	Hash Units	ALU Units	SRAM	TCAM
Utilization	27.3%	31.8%	25.6%	7.9%

Table 7: Hardware resource utilization on the Tofino.

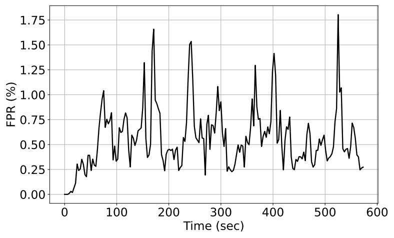

Figure 8: False positive rate fluctuations over ten minutes.

the middle stages. Consequently, NAP applications can coexist on the same hardware target with other applications that balance the resource usage distribution.

5 RELATED WORK

Several network programming languages have been developed for programmable data planes, each with its unique problem scope and approach to addressing resource limitations. Lucid [18] is an event-driven language for implementing low-latency network control on programmable data planes. While it is a concise language, it requires users to define data structures themselves. Lucid's compiler simplifies control flows with static analysis to reduce resource usage mainly in the number of stages, which is orthogonal to our efforts of optimizing data structures under memory constraints.

Marple [17] is a network performance query language backed by a new key-value store primitive, while Sonata [7] is a network telemetry query language that leverages stream processors for scalability. However, neither of them aims to collect network information solely in the programmable data plane. Newton [20] supports network monitoring queries and uses approximate data structures to withstand dynamic network changes, but it lacks a general approach and abstraction of approximate data structures. Furthermore, as network query languages, all of them do not support immediate packet actions based on the result of a query.

P4All [8] extends P4 with parameterized data structures that are optimized by the compiler via integer linear programming. However, granting users the flexibility to define custom data structures and objectives also results in a vast solution space and significantly prolonged compilation times. In contrast, NAP confines users to the data structures offered by our compiler under the dictionary abstraction, resulting in expedited optimization thanks to prior knowledge of resource usage and placement heuristics.

6 CONCLUSION

This paper presents a high-level programming language. With the approximate dictionary abstraction, NAP chooses, sizes, and implements approximate data structures, allowing programmers to write many network applications within 30 LoC, which compile to P4 programs within one second.

ACKNOWLEDGMENTS

This work was supported in part by DARPA grant HR0011-20-C-0160. We also thank the anonymous reviewers of EuroP4'23 for their helpful comments and feedback.

REFERENCES

[1] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. 2022. HeteroSketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 719–741.

[2] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/2934872.2934892

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM* (Hong Kong, China). 99–110.

[4] Andrei Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (11 2003). https://doi.org/10.1080/15427951.2004.10129096

[5] 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 CoNEXT Conference* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3359989.3365408

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

[7] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *ACM SIGCOMM* (Budapest, Hungary). Association for Computing Machinery, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[8] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. 2020. Elastic Switch Programming with P4All *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425933

[9] Intel Tofino1 2021. Intel Tofino chip. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[10] Intel Tofino2 2022. Intel Tofino2 Chip. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[11] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing Key-value Stores with Fast In-network Caching. In *Symposium on Operating Systems Principles (SOSP)*. 121–136.

[12] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-intensive Network Functions on Programmable Switches. In *ACM SIGCOMM*. 90–106.

[13] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*. 334–350.

[14] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*. 101–114.

[15] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap using Switching ASICs. In *ACM SIGCOMM*. 15–28.

[16] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2023. Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1273–1292.

[17] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM* (Los Angeles, CA, USA). Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3098822.3098829

[18] 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). Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/3452296.3472903

[19] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: High-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM*. 207–222.

[20] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2020. Newton: Intent-Driven Network Traffic Monitoring. In *ACM CoNEXT* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3386367.3431298