When P4 Meets Run-to-completion Architecture

Hao Zheng, *State Key Laboratory for Novel Software Technology, Nanjing University, China;* Xin Yan, *Huawei, China;* Wenbo Li, Jiaqi Zheng, and Xiaoliang Wang, *State Key Laboratory for Novel Software Technology, Nanjing University, China;* Qingqing Zhao, Luyou He, Xiaofei Lai, Feng Gao, and Fuguang Huang, *Huawei, China;* Wanchun Dou, Guihai Chen, and Chen Tian, *State Key Laboratory for Novel Software Technology, Nanjing University, China*

https://www.usenix.org/conference/nsdi25/presentation/zheng-hao

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

When P4 Meets Run-to-completion Architecture

Hao Zheng[†], Xin Yan[△], Wenbo Li[†], Jiaqi Zheng[†], Xiaoliang Wang[†], Qingqing Zhao[△],
Luyou He[△], Xiaofei Lai[△], Feng Gao[△], Fuguang Huang[△], Wanchun Dou[†], Guihai Chen[†], Chen Tian[†]
[†] *State Key Laboratory for Novel Software Technology, Nanjing University, China*
[△] *Huawei, China*

Abstract

P4 programmable data planes have significantly accelerated the evolution of various network technologies. Although the P4 language has gained wide acceptance, its further development encounters two obstacles: limited programmability and the cessation of the next-generation Tofino chip. As a hardware manufacturer, we try to address the above dilemmas by opening the P4 programmability of our run-to-completion (RTC) chips. At present, there is no publicly available experience in this field. We introduce P4RTC, a comprehensive consolidation of our experiences applying the P4 language to RTC architecture. P4RTC introduces a new P4 architecture model and a set of beneficial extern constructs to fully leverage the RTC architecture's programmability. Besides, we share the insights we have gained from designing and implementing compilers. We also provide a performance model to facilitate profiling P4RTC's performance on user-customized P4 code. We prototype P4RTC on an RTC chip with 1.2 Tbps bandwidth. Case-oriented evaluation demonstrates that P4RTC can enhance P4 programmability and reduce the burdens of RTC development. The performance model can provide substantial insights into optimizing P4RTC programs.

1 Introduction

P4 programmable data planes have significantly accelerated the evolution of various network technologies, such as in-network computing [1, 2], congestion control [3, 4], load balancing [5], and network measurement [6–8]. As a domain-specific language (DSL) for data-plane programming, P4 has earned widespread recognition in academia and industry for its excellent abstraction. It makes network packet processing more accessible and easier to program. Empirical evidence demonstrates that P4 has become the de facto standard for data plane programmability [9, 10].

Although the P4 language has gained wide acceptance, its further development currently encounters two obstacles. Firstly, its programmability is limited, making implementing many practical designs and algorithms challenging. Currently, P4 is inherently tied to pipeline architectures (*e.g.*, RMT [11]), in which hardware constraints make many algorithms and designs have to make compromises on accuracy or performance [6, 7, 12, 13]. For instance, updating the minimum counter among multiple counters cannot be implemented directly in pipeline architectures since the stage where the minimum counter is located cannot be reaccessed unless recirculation. Secondly, Intel has announced that they will stop developing the next generation of the Tofino series [14], which currently stands as one of the most popular P4 programmable ASICs. To conclude, the P4 community urgently needs a successor hardware platform with Tbps-level throughput and more flexible programmability.

As a hardware manufacturer, we try to overcome the above dilemmas by opening the P4 programmability of our run-to-completion (RTC) chips. RTC is another packet processing architecture significantly different from the pipeline architecture, allowing different packets to have independent processing times. Packets can achieve line-rate forwarding when simple processing is sufficient, while others can be processed below line rate to enable more complex logic. Adapting the P4 language to the RTC architecture carries significant implications for the evolution of the P4 programmable data plane. On the one hand, P4 developers can use RTC devices to implement the functionalities that are difficult to implement in the pipeline architecture. In particular, RTC devices can perform any number of memory read/write operations. Thus, the 'updating the minimum counter' function can easily be implemented. On the other hand, P4 can facilitate the rapid development of the RTC-based data plane. P4 can effectively abstract away the underlying hardware details. Moreover, the recent advancements in the P4 programmable data plane can be easily applied to RTC devices with minimal effort, such as in-network computing [1, 2, 15–17], verification [18–21], packet scheduling [22–24], and formal foundations [25].

At present, there is no publicly available experience in adapting P4 to high-speed RTC devices. Existing extensions to the P4 language [26, 27] focus on reducing the complexity of developing functions and improving resource utilization. They cannot overcome the programmability limitations

of P4 inherent in pipeline architectures. Some efforts (*e.g.*, P4TC [28], P4-DPDK [29], T4P4S [30]) have successfully adapted P4 for CPU platforms. However, they haven't yet extended or optimized P4 for specialized RTC packet processors capable of Tbps throughput. Existing high-level programming languages for high-speed RTC devices are predominantly C-like languages [31]. Their API interfaces are specialized around their hardware and closed to the public community, making it challenging for researchers to understand the underlying architecture and detailed usage.

We propose P4RTC, a comprehensive summary of our experience in applying the P4 language to the RTC architecture. Our contributions can be summarized as follows:

Firstly, we present a new $P4_{16}$ architecture model, which bridges the P4 language and RTC architecture, providing a unified RTC programming abstraction. Based on the architectural model, we introduce a series of novel extern constructs to exploit the flexibility of RTC devices. We show the similarities and differences between P4RTC programming and traditional P4 programming and show how P4RTC can be used to develop functions that are challenging to realize in the pipeline architecture. We also discuss how parallel safety issues are appropriately handled in the P4 code.

Secondly, we share our insights while designing compilers, including table deployments and microcode generation. When deploying P4 tables, we must balance the read/write load among memory banks to improve performance. We develop an integer programming (ILP) model and leverage Profile-Guided Optimization (PGO) [32] strategies to improve the performance of the memory system. When generating microcode, we first transform P4 intermediate representation (IR) to microcode IR rather than directly to microcode instructions. This approach greatly enhances the correctness and efficiency of the generated microcode and makes it easier for the compiler to adapt to different microcode versions.

Thirdly, we provide a performance model to facilitate profiling the performance of P4RTC programs. Due to the resource sharing of the RTC architecture, bottlenecks can occur when many packets simultaneously access the same channel or hardware components. P4 programs no longer have deterministic throughput and delay as they are in the pipeline architecture. The performance model can provide various performance metrics (*e.g.*, pps, latency) to help users identify the bottlenecks of their P4 code under specific traffic patterns.

We prototype P4RTC in Huawei NetEngine 8000 F1A-C routers with 1.2 Tbps bandwidth. The adaptation to higher-throughput RTC chips is ongoing. We conduct several practical use cases. The first case is accurate per-flow monitoring. With the help of P4RTC, the chip can monitor up to 50M concurrent flows at near line rate, and reduce the bandwidth overhead by 86% to 90% compared with an existing Tofino-based solution [33]. The second case demonstrates that P4RTC reduces users' burdens on developing RTC devices. In five projects we examined, P4RTC can reduce lines of code (LOC) by 4.6x to 7.7x compared to microcode programming. This efficiency is comparable to another specialized high-level language developed for the chip. Additionally, using P4RTC, researchers can reproduce the functionalities of existing advances by modifying only 4.3% to 13.0% of the code. In the last case, we show how the performance model guides us in optimizing P4 programs.

This work does not raise any ethical issues.

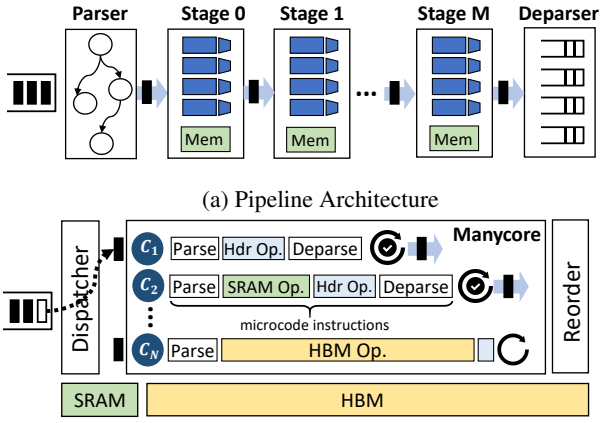


Figure 1: Difference between pipeline and RTC architectures.

2 Background and Motivation

2.1 Architecture Flavors of Dataplane

Packet processing is typically divided into a data plane and a control plane. The data plane, also known as the fast path, is responsible for the processing and forwarding of packets, whereas the control plane handles management logic (*e.g.*, adding routing entries). Developers utilize various hardware platforms, such as CPUs, FPGAs, and ASICs, to implement the data plane for fast packet processing. However, despite significant differences among the hardware platforms, the architecture for the data plane can be fundamentally categorized into pipeline and run-to-completion (RTC) [34].

As shown in Figure 1a, packets orderly pass through several stages in the pipeline architecture. A packet executes part of the processing logic at each stage and then moves to the next stage. Multiple packets can be processed simultaneously in different stages to achieve the 'pipelined' processing. This compact processing is highly efficient when handling high-speed packet arrivals [34]. However, it also introduces some limitations. Firstly, the physical pipeline length is limited in ASICs, which restricts its ability to handle complex logic. Secondly, to ensure the pipeline's efficiency and maximize throughput, resources such as memory are distributed in each stage and cannot be shared between different stages [12].

In the RTC architecture, many cores can independently parse, process, and forward packets. When a packet arrives, a dispatcher assigns the packet to an idle core, and then the

Table 1: Comparison of pipeline and RTC ASICs

Feature	Pipeline ASIC	RTC ASIC
Throughput	(Serial) High	(Parallel) Moderate
Latency	O(100) ns	O(100) ns - ?
Memory Capacity	10's MB	10's - 1000's MB
Logic Length	Limited	Unlimited

packet completes the entire processing on that core. This design allows different packets to have independent processing time, which overcomes the limitations introduced by the pipeline architecture. In particular, packets can perform unlimited read and write operations to a shared memory to implement complex algorithms at the expense of increasing processing time. As for the forwarding performance, RTC devices adopt a manycore architecture to improve the overall throughput to the Tbps level [31]. However, this can lead to memory conflicts and packet disorders.

Table 1 provides an overview of the distinct features of the two architectures. The pipeline architecture typically achieves high throughput under serial processing, while the RTC architecture needs to increase the number of forwarding cores (*i.e.*, parallelism) to improve throughput. Regarding latency, the pipeline architecture generally exhibits deterministic low latency in forwarding, whereas RTC devices' latency depends on the specific processing logic of the packets. As for memory capacity, RTC supports large-capacity memory such as high-bandwidth memory (HBM). Finally, both architectures have the potential for programmability.

2.2 Dataplane Programmability and P4

Reconfigurable Match-action Table (RMT [11]) is a famous programmable switch architecture. It is a typical pipeline architecture consisting of several reconfigurable match-action units (MAU). Developers program the data-plane logic by reconfiguring the structure of tables within the MAUs, including the match-phase key and the action-phase instructions. However, as mentioned earlier, its programmability is constrained, including the limited number of MAU stages, limited capacity and isolated memory. As another crucial packet processing architecture, some RTC devices [31] are also programmable by reloading microcode into chips. Regarding programmability, RTC devices offer more flexibility because they remove the constraints imposed by physical stages (*i.e.*, fixed lengths and isolated memory). This significantly bridges the gap left by the pipeline architecture.

For chip users, configuring the pipeline hardware or developing microcode can be laborious. P4 is a high-level language for data-plane programming, which was initially proposed in 2014 [35]. The P4 data plane consists of a series of programmable blocks, mainly including `parser`, `ingress`, `egress`, and `deparser`. These programmable blocks compose a logical pipeline named a top-level `package` or pipeline package. As an instance, the `main` object in Figure 2 is a

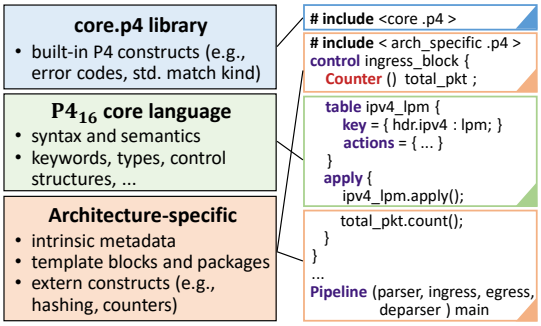


Figure 2: P4$_{16}$ language structures

pipeline package. Developers can customize programmable blocks within the pipeline package. P4 enables developers to implement diverse packet processing logic using only general-purpose operations and table look-ups. More importantly, P4 provides a general packet processing abstraction to shield developers from the underlying hardware details.

However, the original P4$_{14}$ language is complex, mainly attributed to the integration of architecture-specific features into the language, making it hard to adapt to other architectural models [36]. P4$_{16}$, as an improved version of P4$_{14}$, separates the architecture-specific features from the P4 core language. As shown in Figure 2, the basic syntax, semantics, and keywords (*e.g.*, `table`, `apply`) are defined in P4$_{16}$ core language. In contrast, the programmable blocks, extern constructs (*e.g.*, `Counter`), and pipeline packages are architecture-specific. P4$_{16}$ makes the P4 language relatively stable, leaving the architecture-specific parts to be implemented by target manufacturers. However, all existing P4 architecture models are based on the pipeline architecture, such as the PISA [37] used in P4$_{14}$, Portable Switch Architecture (PSA) [36], and Tofino native architecture (TNA) [38] introduced after P4$_{16}$. The adaptation of P4 to the RTC architecture remains unexplored, leaving a gap in the current research community. As a result, when researchers consider utilizing the P4 programmable data plane for designing new network features, they commonly regard the pipeline architecture as their primary target.

Providing a publicly available RTC programming model based on P4 is highly essential. Firstly, introducing the RTC architecture will enhance the programmability of the P4 programmable data plane. As discussed earlier, the pipeline architecture sacrifices programmability to enhance overall forwarding efficiency, which forces many algorithm implementations to make compromises. For instance, some systems execute their algorithm through recirculations, albeit at the cost of reduced bandwidth [4, 39]. Others [6, 40] opt to compromise on the design of their algorithms, implementing an approximate version instead, which inevitably results in a decrease in the algorithm's accuracy. Fortunately, the RTC architecture can make up this aspect, thanks to their unlimited logical length and large-capacity shared memory. With this

flexibility, P4 programmers can implement network features previously unachievable in the pipeline architecture. On the other hand, P4 can facilitate the rapid development of the RTC-based data plane. P4 abstracts away the tedious hardware details for users, while ongoing support for P4 allows existing advancements from the P4 community to remain valuable within the RTC architecture.

3 P4RTC Overview

As shown in Figure 3, the workflow of P4RTC consists of four stages: P4 programs, compilation, firmware, and targets.

Firstly, users define their customized data plane and control plane programs. In the compilation stage, a P4 compiler transforms the P4 code into microcode instructions, and a microcode compiler further generates a binary file according to it. The deployment of P4 tables is also completed at this stage. The binary file and configuration files generated during the compilation stage are collectively called the chip's firmware. Finally, the firmware is loaded onto the target hardware.

Given that both P4 and our RTC chip have reached mature stages individually, we strive to maximize component reuse during the implementation of P4RTC. As shown in Figure 3, the grey components are reused components. Specifically, P4 has public specifications and open-source compilers [41]. The chip used is not a brand-new design but was chosen from our existing RTC chips, previously used for fixed-function routers. Therefore, some legacy components exist, such as the microcode compiler and software development kit (SDK). However, we still encounter the following three challenges:

How to extend P4 to fully leverage the flexibility of RTC without altering P4$_{16}$ core language? Since the P4 language was initially designed around the pipeline architecture, directly referencing the existing architecture models cannot fully utilize the flexible programmability of RTC devices. To this end, we propose a new P4 architecture model for the RTC architecture. Under this architecture model, we provide a set of novel extern constructs (in `RTC.p4`) rather than modify the syntax and semantics of P4. We will introduce the new P4 architecture model and extern constructs in §4.

How does the compiler map the P4 language to the RTC architecture? Due to architectural differences in the hardware, the compiler needs to be designed explicitly for the RTC architecture, especially in table deployments and microcode generation. We implement a new backend for the open-source P4C compiler to generate RTC-specific microcode and table structures. We will introduce the novel issues in the table deployments of the RTC architecture and the experience in the microcode generation process in §5.

How do we verify the performance of the chip after loading customized P4 code? Unlike deterministic performance in the pipeline architecture, the flexibility of the RTC architecture makes it easy to introduce poor performance code. Conducting real machine experiments takes a lot of time and effort. It involves setting up connections, generating traffic, and collecting and analyzing results. More importantly, it is hard to accurately capture the performance details of the chip's internal components. To this end, we introduce a performance model to help users verify their P4 code in §6.

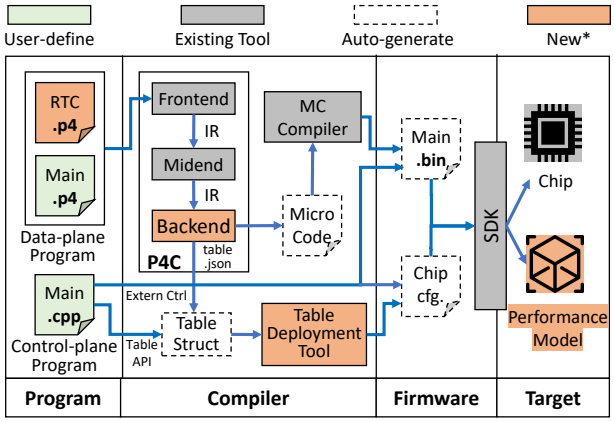


Figure 3: P4RTC system overview. The newly introduced components to address the challenges are colored orange.

4 P4RTC Programming

4.1 New P4 Architecture Model

Figure 4 illustrates the new P4 architecture model introduced by P4RTC. Considering a regular packet forwarding path, a packet enters the chip through the RX MAC and is dispatched to an idle core, where each core can be understood as a thread. Once a packet is handled by a core, all operational logic is executed on that core, where P4 programs define how to process and forward the packet. After processing, the packet is sent to a reorder engine for flow-level sequencing. Then, the packet is sent to the output queue of the destination port through the traffic manager (TM). After leaving the TM, the packet bypasses egress processing by default and is directly sent to TX MAC. If egress processing is necessary, it will again be assigned to a core. In general, the processing logic of the cores is the programmable part of the RTC architecture, while the other parts are non-programmable.

When programming RTC devices with the P4 language, the programmable blocks that users must write reserve those in the pipeline architecture, including the parser, ingress/egress control flow, and deparser. In other words, in the perspective of packets, packets are still processed by a logical pipeline composed of the traditional P4 programmable blocks. At the same time, the user's programming habits are also preserved. However, the concept of 'pipeline' in P4RTC has the following three main differences.

Many-core programming model. Packets are no longer serially processed by a single pipeline of physical stages. In contrast, there are many cores in RTC devices, and each packet is processed by an independent 'logical pipeline' operating on a dedicated core. Users define a pipeline package consisting of P4 programmable blocks, and then the cores run this package to process the packets in parallel. As shown

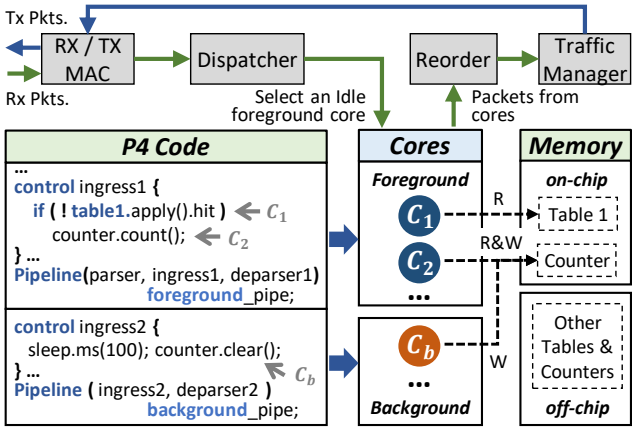


Figure 4: P4 programming abstraction for RTC architectures

in Figure 4, both the core C_1 and the core C_2 are running the *foreground_pipe* package defined in the P4 code. While C_1 is accessing `table 1`, C_2 updates a counter.

Unlimited logical length. In the RTC architecture, packet processing is performed by a dedicated core executing microcode instructions. The logical length is no longer constrained by the number of physical stages, as is the case in the pipeline architecture. This feature dramatically enhances chip programmability, but at the same time, it can easily lead to inefficient code, which degrades overall throughput.

Shared memory subsystem. Packets are no longer sequentially processed by multiple physical stages. Thus there is no need to access separate memory on these stages. Multiple cores can simultaneously access a shared memory system through on/off-chip memory controllers and crossbars. Given the shared nature of the memory, any memory location (*e.g.*, match-action tables, counters, and registers) can be accessed multiple times during the packet processing. As an instance, both C_1 and C_2 can access the counter extern in Figure 4.

4.2 Extensions for Pipeline Package

In the RTC architecture, the running instructions of each core can be defined separately. To provide a simplified view of P4 programming, the pipeline packages are grouped into two roles: foreground pipeline and background pipeline. By default, most cores run the foreground pipeline package, responsible for parsing, processing, and forwarding received packets. The other cores step back into the background. As shown in Figure 4, foreground cores C_1 and C_2 are running the P4 code in the 'foregound_pipe'. C_b is running the 'backgound_pipe' independently as a background core, which resets the counter every 100 ms.

A core running the background pipeline is stuck in an infinite loop of ingress $\rightarrow$ deparser $\rightarrow$ ingress $\rightarrow \cdots$. The background pipeline has no parser block since it does not require packet reception. However, it does not mean that background pipelines cannot process packets. Like standard P4 programming, users can use a `setValid()` method to enable protocol headers within the ingress block. When the control flow reaches the deparser, protocol headers marked as valid will be emitted. Therefore, the background pipeline can also generate customized packets.

To further exploit the programmability of the background pipeline, we introduce the following P4 extern constructs:

```
extern Foreach{
  void apply(in bit<32> start,in bit<32> end);
}
extern void break();
extern void continue();
extern Sleep{
  void us(in bit<32> value);
  void ms(in bit<32> value);
}
```

The `Foreach()` extern allows iteratively access to table entries by binding with a table. It can also be bound to a virtual table to implement a generic range for loop. The reason why we do not introduce a 'for loop' syntax like C/C++ is to retain the P4 core language, which is a crucial design philosophy of $P4_{16}$ for its target-independent feature. The `Sleep()` extern controls the activation frequency of background cores. The following example shows how to use the *Foreach* extern:

```
control back_ingress{
  action __for_act0(KEY_T k, DATA_T data)
  { /* loop body */ }
  @tbl("t")
  @methods(act0="__for_act0")
  Foreach() foreach_t;
  Sleep() sleep;
  apply{
    sleep.ms(10);
    /* setValid() related headers. */
    foreach_t.apply();
  }
}
control back_deparser {
  apply { pkt.emit(hdr); }
}
Pipeline(back_ingress, back_deparser) back_pipe;
```

In the above code, the `Foreach` extern binds a table called `t` and a loop body (*i.e.*, `__for_act0`). The core running the `back_pipe` will traverse each row of `t`, read the match key and action data of that row, and pass them to the loop body to perform the user-customized operations. Once the table traversal is complete, the control flow is transferred from `back_ingress` to `back_deparser`, and the `back_deparser` block sends a packet if any headers are valid. The `Sleep` extern can control the loop frequency.

It seems that the background pipeline is similar to traditional control plane threads. The main differences are that (i) it runs directly on the data plane and has a shorter control loop, and (ii) it has better performance scalability.

Inter-pipeline communication. To better assist in functional decoupling between different pipeline roles, we propose a `Queue` extern construct.

```
extern Queue<T> {
    Queue(bit <32> depth);
    bool push(in T value);
    bool pop(out T value);
    bool popWait(out T value);
}
```

The pipelines running on different cores can implement message push, pop, and a blocking pop through a parallel-safe queue extern. For example, the `Queue` extern can be used to implement messages passing between the foreground pipeline and the background pipeline. In the measurement project outlined in §7.1, we implement a background pipeline to periodically age out expired flow entries (*i.e.*, those that need to be reported to an analyzer). Nevertheless, employing a linear scan of the flow table proves inefficient, as the flow entries accessed by the background pipeline frequently do not meet the aging criteria. We adopt a smart aging technology to improve aging performance. When a foreground pipeline receives the FIN or RST flagged packet, it pushes the corresponding flow information into a `Queue` extern. The background pipeline just needs to pop flow IDs from the queue and directly complete the aging of the flows, which can significantly improve the aging performance. The corresponding P4RTC pseudocode can be found in Appendix A.

4.3 Extensions for Match-action Table

In P4RTC, we expand the types of match-action tables. According to the deployment location, tables can be divided into on-chip tables and off-chip tables. The on-chip tables have lower accessing latency, but the capacity is only a few tens of MB. In contrast, off-chip tables have GB-level capacity, but the access latency is an order of magnitude higher.

According to the memory layout, tables can be classified into content addressing tables (CAT) and linear addressing tables (LAT). The CAT tables perform exact matching and longest prefix matching (LPM) by storing keys in SRAM or specialized memories like content-addressable memory (CAM). In comparison, the LAT tables directly utilize the key as the memory index and then perform the action phase based on the action ID and action data stored in the memory address. LAT tables are more efficient than CAT tables, while it is limited by the capacity of the address space. They are commonly used for short-bit-width matching operations, such as Port-VLAN mapping.

The default definition of the P4 table is an on-chip CAT table. Users can use annotations to change the attributes of the table, including deploy locations, table layouts, and bank group size in the following format:

```
@linear
@offchip(x4)
table offchip_linear { ... }
```

The first annotation, `@linear`, indicates that a LAT table is required instead of CAT table. Another annotation, `@offchip`, signifies that the table needs to be deployed in off-chip memory to accommodate a larger table size. Additionally, users can specify the size of the bank group in the format `@offchip(x<group_size>)`, which improves memory read performance for off-chip tables.

In the RTC architecture, packets can operate table entries directly in the data plane, both for CAT tables and LAT tables. Compared with existing pipeline architecture that can only add or delete CAT entries from the control plane, the in-data-plane table operations have a shorter control loop, thus reducing the delay from milliseconds to hundreds of nanoseconds. To utilize this feature, we introduce a new P4 extern construct:

```
extern TableOperation {
  TableOperation();
  bool entryDel<KEY_T>(in KEY_T k);
  bool entryAdd<KEY_T, DATA_T>(in KEY_T k,
                               in bit<8> act,
                               in DATA_T data);
  bool entryGet<KEY_T, DATA_T>(in KEY_T k,
                               out bit<8> act,
                               out DATA_T data);
}
```

Table operations enable much of the logic that needs to be initially executed on the control plane to be efficiently offloaded to the data plane. A typical use case is to use the on-chip memory as a cache for off-chip big tables. We can dynamically add frequently hit entries to the on-chip memory. The code below shows an implementation example:

```
control foreground_ingress{
  table onchip_cache { ... }
  @offchip
  table offchip_tbl  { ... }
  apply{ /* Only apply the off-chip table
    when a cache miss occurs. */
    if(onchip_cache.apply().hit == false)
      offchip_tbl.apply();
  }
}
control background_ingress{
  @tbl(onchip_cache)
  TableOperation() cache_op;
  @tbl(offchip_tbl)
  TableOperation() tbl_op;
  apply{
    /* Find a heavy key, denoted as 'hk' */
    bool re = cache_op.entryAdd(hk, act, data);
    if(re == false)
      { /* The cache is full. Make eviction. */}
  }
}
```

In the above code, we implemented a simple caching strategy for an off-chip large table named `offchip_tbl`. The `foreground_ingress` only accesses `offchip_tbl` in case a cache miss happens. The `background_ingress`, which is packaged in a background pipeline, periodically loads hot entries into the `onchip_cache` through the `entryAdd()` table operation. Note that we omit the selection of the heavy key and the cache eviction strategy in the code. Through the `Foreach` extern, users can customize their heavy-key selection and idle-key eviction strategies.

Furthermore, we introduce a `lastRowIndex()` extern method. If the `lastRowIndex()` method is called after applying table or executing table operations, it returns the row id of the operated table entry. An important use of `lastRowIndex()` is attaching counters and registers with match-action tables, thereby constructing some advanced data structures. For example, suppose we have an exact matching table with size n and a counter array with size $2n$. With the `lastRowIndex()` extern, the counters can serve as a one-to-two supplement for the exact matching table, which

can implement packet&byte counters for each table entry.

```
control foreground_ingress{
  table tbl {... size=64}
  Counter() cnt(128);
  apply{
    table_1.apply();
    bit<32> row_index  = lastRowIndex();
    bit<32> pkt_index  = row_index << 2;
    bit<32> byte_index = (row_index << 2) + 1;
    cnt.count(pkt_index, 1);
    cnt.count(byte_index, getPktLength());
  }
}
```

4.4 Parallel Safety

A non-negligible concern is the handling of parallel safety of the RTC architecture. Most data-plane operations are stateless operations, which will have no parallel contention issues. However, some stateful operations involve modifying the device state. If not handled, it will lead to unpredictable errors.

The first type of stateful object is the `Counter` extern. The counter operations including `count`, `get`, and `set`, are implemented as atomic operations in hardware. The second type of stateful operation is the `TableOperation` extern that makes addition and deletion of flow tables. These operations involve inserting items into a hashing structure, such as cuckoo hashing [42] or d-left hashing [43], which incurs parallel issues when handling hash conflicts. Fortunately, users can regard the entire operation as an atomic transaction because efficient hardware locks guarantee parallel security.

However, when the above operations are combined to implement read-modify-write operations, it can lead to race conditions if not correctly synchronized. The key to ensuring parallel safety in read-modify-write operations is to prevent all other cores from initiating the same operation until the first core has completed its task. In P4RTC, we provide a `Lock` extern construct to solve this problem.

```
extern Lock {
  Lock(bit<32> size);
  bool acquire(in bit<32> idx, in bool blocking);
  bool release(in bit<32> idx);
}
```

The underlying of the `Lock` extern is the test-and-swap (TAS) atomic instruction. A core obtains a lock that can either be blocking or non-blocking. The distinction is whether the core chooses to spin and wait if it cannot acquire the lock immediately. A `Lock` extern can contain multiple lock instances, and users can implement row-level locking by utilizing the `lastRowIndex()` extern method mentioned before.

5 P4RTC Compilation

This section will discuss our experience in developing the P4RTC compiler, focusing on two main aspects: table deployments and generating microcode instructions.

5.1 Table Deployments

Table deployment entails strategically allocating P4-defined tables (*i.e.*, match-action tables, counters, registers) to the chip's physical memory resources. In the RTC architecture, there is no need to consider table dependencies as in a pipeline architecture; only capacity constraints and performance optimization issues need to be addressed.

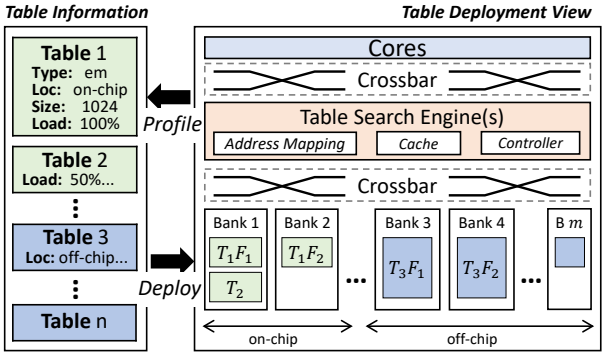


Figure 5: Table deployment view in P4RTC

Problem Constraints and Optimization Objective. The table deployment problem in the RTC architecture can be modeled as a variant of the bin packing problem (BPP), where the bins are the memory banks, and the items are the P4-defined tables. There are two main differences from the standard BPP. Firstly, in the shared memory subsystem, read and write requests caused by table matching and `TableOperation` extern will be queued in the channel of corresponding memory banks. Specific memory banks may suffer congestion due to simultaneous access by multiple cores, and the negative effect is more evident for off-chip deployed tables with higher latency. Therefore, the optimization goal is not to minimize the number of memory banks used but to minimize the maximum load among the memory banks to optimize the performance of the memory subsystem [1].

Secondly, P4RTC allows table fragmentation to utilize memory capacity fully. Specifically, we can fragment a table to satisfy capacity constraints or to reduce the load the table imposes on a single bank. However, each table fragmentation occupies an address mapping entry in table search engines (TSEs [2]), and the number of mapping entries available is limited. Consequently, this introduces a constraint that the quantity of table shards must be less than the number of supported mapping entries.

Based on the above considerations, we use an integer linear programming (ILP) model to solve the problem. Given n tables and m memory banks, each table i has a size of s_i, where $i = 1, 2, \ldots, n$. For each memory bank j, it has a capacity of c_j, where $j = 1, 2, \ldots, m$. We define an integer variable x_{ij} to represent the size of the fragment of the table i put in the memory bank j. We consider this variable as an integer (included between 0 and s_i). We also define a decision variable y_{ij} that equals 1 if the fragment of the table i is deployed

[1] Note that the load balancing of memory requests involves more hardware components (e.g., table search engines). Our experience is that as long as the load balancing of memory banks is done well, it is straightforward to handle the balancing of other upper-layer components.

[2] Table search engines are specialized memory controllers optimized for table operations, efficiently managing table search, read, and write requests.

on memory bank j, and 0 otherwise. Then, the total number of fragments of table i can be expressed as $\sum_{j=1}^{m} y_{ij}$. The following ILP model formalizes the table deployment problem:

$$\text{minimize} \quad \max_{j\in\{1,\ldots,m\}} \sum_{i=1}^{n} x_{ij} \cdot l_i \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{m} x_{ij} = s_i, \quad \forall i \in \{1,\ldots,n\} \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} \le c_j, \quad \forall j \in \{1,\ldots,m\} \tag{3}$$

$$\frac{x_{ij}}{s_i} \le y_{ij}, \quad \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{4}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \le T \tag{5}$$

$$x_{ij} \in \mathbb{N}, \ \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{6}$$

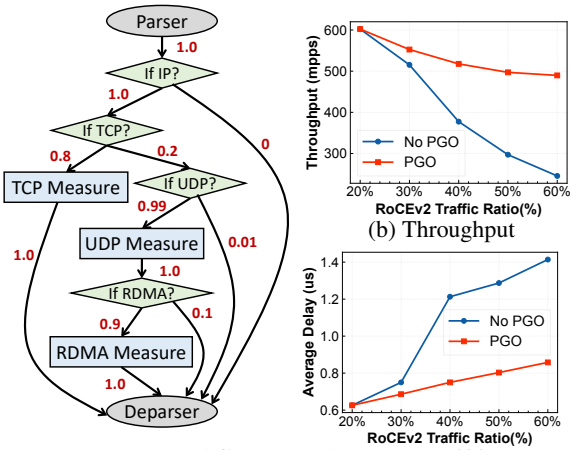
$$y_{ij} \in \{0,1\}, \ \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{7}$$

Here, l_i denotes the load incurred by the unit size of table i, while T denotes the maximum number of available mapping entries. The optimization objective (Equation (1)) is to minimize the maximum access frequency among all memory banks. Equation (2) is the table assignment constraint, ensuring that each table is fully deployed. Equation (3) guarantees that the capacities of the memory banks are not exceeded. Equation (4) mandates that the variable y_{ij} must be equal to 1 as soon as the variable x_{ij} is strictly greater than 0. Finally, Equation (5) ensures that the maximum number of mapping entries does not exceed.

Profile-guided optimization on table deployments. The above model relies on load information for each table. However, table access's read/write frequency is determined by runtime traffic and specific table entries. We employ a profile-guided optimization (PGO) compilation strategy. Initially, we adopt a profile where each branch evenly shares the load of its parent node. Then, after running the switch, a new table load information can be obtained [3], and we use the new profile to solve the table deployment problem again. Empirically, we perform PGO 2-3 times in the early stages of deployment to gradually adapt to the traffic of the new environment. In the subsequent stages, we will only proceed with a new round of PGO iteration if there is a significant change in the workload and it stabilizes in the new state.

The PGO's performance maintenance and enhancement are especially noticeable in scenarios with significant workload changes. For instance, within our traffic measurement project, we employ distinct off-chip tables to measure TCP, UDP, and RoCEv2 traffic, enabling the collection of protocol-specific metrics. The DAG graph of the program is shown in Figure 6a. Initially, TCP traffic predominated; however, with the gradual integration of RDMA technology across the network, RoCEv2 traffic has steadily gained prominence over the past years, as it also occurs in Microsoft Azure Storage [44]. We are comparing the performance of two strategies: one utilizing PGO optimization and the other not

[3]The compiler will automatically insert counter instrumentation.



(a) Program DAG (b) Throughput (c) Latency (400 Mpps)

Figure 6: Effect of PGO on performance.

using PGO. The PGO optimization strategy employs the optimal configuration based on the latest workload, whereas the non-PGO strategy consistently utilizes the initial configuration with RDMA traffic at 20%. As shown in Figure 6, as the proportion of RoCEv2 traffic increases to 60%, the non-PGO strategy's throughput drops by 64.7%, and its latency increases by 2.3x. In contrast, the performance of the PGO strategy remains good, with a slight performance degradation as RoCEv2 traffic needs to access more off-chip tables.

5.2 Microcode Generation

P4 is a high-level language that cannot be run directly on chips. Microcode are the closest language to the RTC chip's machine code, usually built from a specific instruction set designed by manufacturers. We must implement a compiler to generate the microcode instructions from P4 code. We reuse the open-source P4C project [41] but rewrite the compiler's backend. Due to space constraints in the paper, we provide an overview of the compilation process and our experiences. A detailed example of compiling P4 to microcode and intermediate representations (IR) is elaborated in Appendix B.

Define Microcode IR. Lowering P4 IR (*i.e.*, generated by P4C) directly into the microcode would increase the coupling between the compiler and microcode, making the compilation process difficult to control. We introduce a Microcode IR, enabling detailed control, optimization, and validation, considering the specific characteristics and constraints of the target hardware, as we will discuss further. Figure 7 shows part of Microcode IR definition for the `Foreach` extern:

Transform the P4 IR to Microcode IR. This is the most complex step in the compilation process. An example of the `Foreach` extern translation is shown in Figure 8.

1. *Align P4 IR with Microcode IR.* Specific Microcode IR nodes can be directly mapped from the equivalent nodes in P4 IR, such as assignment and logical operations shown in the right part of Figure 7. This step identifies the corresponding operations and control flow constructs within

```
/*McIR definition diff.
  from P4IR */
class Branch:Statement;
class Label:Expression;
class LabelStmt
          :Statement;
class EntryGet
          :Statement;
...
```

```
/*McIR definition equiv.
  to P4IR */
class AssignmentStmt
          :Statement;
class IfStatement
          :Statement;
class Lss
          :Expression;
...
```

Figure 7: Microcode IR Definition

the Microcode IR that align with those in the P4 IR. During this process, the compiler must carefully handle type translation, including endianness, padding, and bit width.

2. *Implement P4-specific and architecture-specific constructs.* Handle complex P4 constructs such as match-action tables by mapping them to corresponding Microcode IR nodes. New extern constructs introduced by P4RTC are implemented in this step. As shown in Figure 8, the `Foreach` extern is mapped to an assembly-like loop control flow, including a series of label statements, jump instructions (*i.e.*, Branch), and loop body. This process reads relevant information from multiple parts of the P4 IR based on the desired loop structures in microcode.
3. *Manage Dependencies.* Ensure that the semantic properties and constraints of the P4 program are preserved in the Microcode IR. Consider the dependencies between P4 IR operations and ensure they are correctly accounted for in the Microcode IR. This includes preserving data dependencies, control flow dependencies, and ensuring the proper ordering of instructions in the microcode to avoid hazards or incorrect behavior.
4. *Validate and Verify.* Ensure the correctness and integrity of the transformed Microcode IR. This includes maintaining correct packet processing behavior, adherence to P4 language rules, compliance with micro-architectural restrictions or limitations, and checking for potential errors and inconsistencies.

Emit Microcode Instruction. After completing the previous phases, the lowered Microcode IR can be directly mapped to the microcode instructions (please see Appendix B). The more details are lowered in the Microcode IR level, the less complexity there is in the emission stage, resulting in fewer introduced bugs. Besides, when facing different versions of microcode, Microcode IR allows developers to reuse existing compilation logic, especially target-independent compilation passes, such as assignment and logical operations.

6 Performance Model

In the RTC architecture, P4 codes and actual workloads significantly impact the chip performance. Therefore, strict performance testing must be conducted on P4 programs before deployment. Besides standard functionality models like BMv2 [45], we introduce a performance model for P4RTC to explore potential bottlenecks and optimize users' programs.

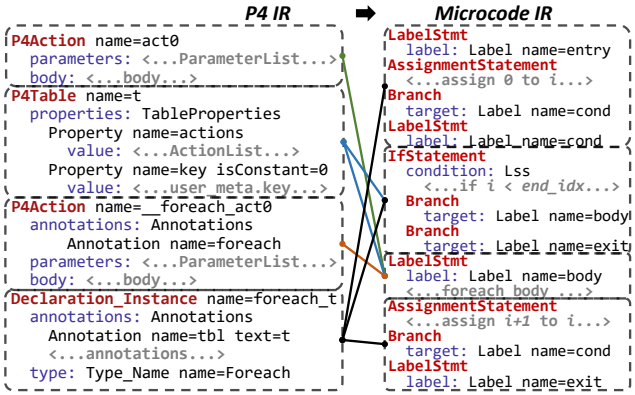


Figure 8: An example of P4 IR to Microcode IR

In general, performance models are utilized to evaluate metrics such as the throughput and latency of a chip under specific programs and workloads [46, 47]. We build a system-level model utilizing the Electronic System Level (ESL) methodology, leveraging the robust capabilities of the SystemC library [48]. We model main hardware components, encompassing cores, the traffic manager (TM), TSEs, the memory subsystem (MSS), and their interconnections (*e.g.*, FIFOs and back-pressure mechanisms). The model employs a global clock (*i.e.*, `sc_clock` in SystemC) to synchronize the chip components, define their operational frequencies, govern instruction execution, and regulate packet rates. As a result, the model can simulate cycle-level behaviors across a range of packet processing scenarios, enabling a more granular real-time depiction of each chip component's status.

As shown in Figure 9, our performance model uses four steps to obtain the performance metrics:

Step 1. Given a P4 program and a specific workload (*i.e.*, a packet trace), we perform functional simulations [45, 49] to process packets, record the microcode paths that each packet traverses, and then save the paths to a *codepath* file. For instance, the sequence `Parser` → `TCP Measure` → `Deparser` in Figure 6a represents an execution path. To realize the system-level simulation, we must also model the hardware behaviors. Specifically, when applying a P4 table, packets access various table entries, thus generating different microcode instructions that interact with multiple memory banks. However, for large P4 projects, recording all different microcodes for an extensive number of packets may result in the *codepath* file becoming excessively large. To deal with these scenarios, we use a probabilistic approach to construct the *codepath* file. We summarize all microcode paths in the *codepath* file as multiple *branches* and *sub-branches* of microcodes, each bearing distinct probabilities. The below example illustrates a sample *codepath* file:

```
<codepath name="demo" bw="1000Mpps">
  <microcode percentage="90%">
    <!-- microcode branch 1 -->  ...
    <io type="read" table="IPv4", tse="0", ...>
      <!-- sub-branch -->
```

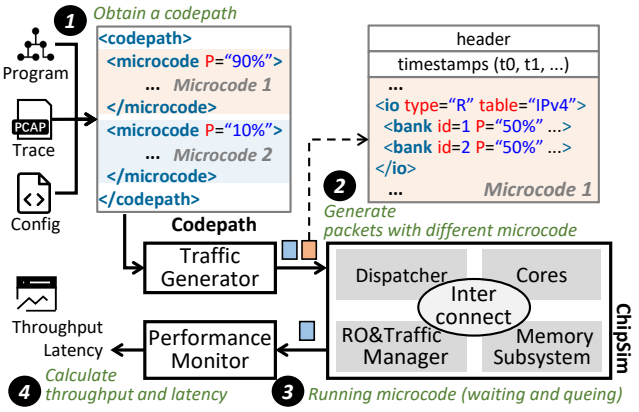


Figure 9: Workflow of performance model

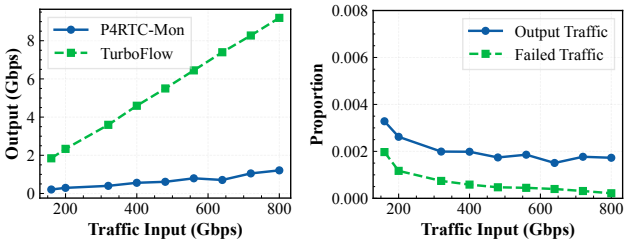
```
      <bank id="31" percentage="50%", .../>
      <bank id="32" percentage="50%", .../>
    </io>
    <action>
      <!-- sub-branch -->
      <alu name="l3_fwd" percentage="90%", .../>
      <alu name="drop" percentage="10%", .../>
    </action> ...
  </microcode>
  <microcode percentage="10%">
    <!-- microcode branch 2 --> ...
  </microcode>
</codepath>
```

There are two microcode *branches*, each with several *sub-branches*. The selection of the *branches* and *sub-branches* is performed dynamically in the following steps.

Step 2. After obtaining a *codepath* file, we upload it to a traffic generator. The traffic generator then interprets the *codepath* and generates traffic at a specified rate. Specifically, the traffic generator creates packets based on the probability of each microcode *branch*. Each packet includes a reference to the relevant microcode instructions, an identifier, and a metadata space for storing timestamps, which are used to calculate the latency experienced in each chip component.

Step 3. The chip simulator receives packets and navigates the microcode instructions. For each instruction, the simulator only emulates the performance behavior rather than executing it precisely. Take an IO instruction as an example: it will generate specific IO requests for the memory subsystem and emulate behaviors, including addressing and queuing a particular memory bank. Each behavior costs specific cycles, consistent with the chip's microarchitecture [50]. When encountering a *sub-branch*, the simulator dynamically employs a probabilistic process to execute the selected instruction, such as bank selection. When a component is over-stressed, a back-pressure signal is sent to upstream components, triggering chip-wide performance impacts.

Step 4. A performance monitor receives the packet output from the chip simulator and summarizes various performance metrics, including throughput and latency. Moreover, the performance monitor can acquire load or utilization information on each hardware component by inserting monitoring probes



(a) Bandwidth Cost (b) Traffic Proportion

Figure 10: Accurate Per-flow Monitoring

(*i.e.*, codes for monitoring) into the chip simulator.

In §7.3, we demonstrate how the performance model can guide performance optimization and assess its accuracy by comparing its outputs with those of a hardware chip.

7 Case Study

We prototype P4RTC on a 1.2-Tbps RTC chip with 8 GB high-bandwidth memory (HBM), used in Huawei NetEngine 8000 F1A-C (NE8000F1AC) routers. The adaptation to higher-throughput chips with the same architecture is ongoing. We evaluate P4RTC through several use cases and summarize the following findings:

P4RTC benefits P4 programmability. It incorporates RTC architectures into the P4 community, enabling previously unsupported functions in P4. A comprehensive case on accurate per-flow monitoring demonstrates that P4RTC can monitor up to 50 million concurrent flows at a near line rate. When the input traffic surpasses 800 Gbps, the report traffic sent from the data plane to a collector occupies less than 0.06% of the total input bandwidth (see case 1).

P4RTC benefits RTC-based devices. Besides existing C-like high-level languages, P4RTC offers another efficient method for data plane development. It reduces the lines of code (LOC) by 4.6x-7.7x compared with microcode programming. Besides, existing P4 functions can be migrated to P4RTC with minimal effort (see case 2).

The performance model benefits P4RTC optimization. It details the performance impact of different P4 codes on the chip under specific traffic inputs. In case3, we employ the performance model iteratively to identify system bottlenecks and implement targeted optimizations. Finally, we enhance the throughput to 14.4 times that of the baseline program.

7.1 Case#1: Accurate Per-flow Monitoring

Accurate per-flow monitoring is helpful for network performance analysis since it completely records flow-level traffic information. Due to the limitation of memory capacity and programmability, the task is challenging to implement on existing pipeline ASICs. Existing efforts employ a sampling strategy to monitor packets with general-purpose servers [51]. Nevertheless, this sampling strategy can lead to imprecise statistics and result in substantial information loss.

With P4RTC, implementing accurate per-flow monitoring

becomes straightforward and flexible. Specifically, we can define an off-chip big table as a flow measuring table (FMT) and the matching key as a flow key (*e.g.*, 5-tuple). Then, some counters and registers are defined to record the size, times-tamps, and other flow attributes. When a packet arrives in the foreground pipeline, it first tries to match a table entry. If a table miss occurs, we use `TableOperation` to add a new flow entry for recording. On the other hand, the FMT needs to age in time to make room for new flows. We complete the aging function by combining the foreground pipeline and background pipeline. In the foreground pipeline, we identify end-of-flow signals such as TCP's FIN and RST flags. In the background pipeline, we use the `Foreach` extern to traverse table entries, and we also use the `Queue` extern to accelerate the performance (see Appendix A). If a flow record is marked as finished or there is no packet access after a certain timeout, the flow will be reported and deleted from the FMT. This background pipeline can be bound to multiple cores to improve aging performance, where each core is responsible for aging a portion of the FMT.

We implement the above function with P4RTC. We connect a NE8000F1AC device to two ToR switches of a storage cluster (with a mix of client traffic and backend traffic) and mirror the truncated upstream packets to the device. With 4 GB of memory, it can measure up to 50M concurrent flows. It is lower than ideal (*i.e.*, 4 GB / the size of a flow record) due to hardware details such as memory alignment and redundancy of hash tables to reduce conflicts. We compare the performance with TurboFlow [33], a state-of-the-art accurate per-flow monitoring system based on Tofino. Turboflow evicts older flow records to the control plane when a hash collision occurs. In contrast, P4RTC can use the *TableOperation* extern to add flow records to an exact match table, automatically handling hash conflicts in the data plane. Therefore, it can continue to measure flows regardless of conflicts, reducing the additional bandwidth overhead caused by hash conflicts. As shown Figure 10a, our solution reduced the bandwidth overhead for sending measurement data by 86% to 90%. Note that packets will fail when multiple packets of a flow add entries simultaneously (*i.e.*, there is a row lock when adding table entries). In Figure 10b, we find that the bandwidth occupied by the failed packets is less than 0.2% of the input traffic. This is because packet failures occur only during the creation of flow entries due to lock contention. Once the table entries for the flows are established, subsequent packet counter updates proceed normally. Since the time taken to build entries in the data plane is short, the number of failed packets is minimal.

7.2 Case#2: Reduce RTC Developing Burden

Table 2 shows the comparison of the number of P4 code lines required for developing functions such as accurate flow monitoring in case 1 and SpaceSaving algorithm in Appendix C. We find that P4 can reduce the LOC development by 4.6x-7.7x compared with microcode programming in the five projects. This efficiency is comparable to NPC++, a C-like programming language developed for the chip. The NPC++-related projects are still under development. We estimate the LOC needed to replicate the same functionality using pseudo code written in NPC++ syntax. NPC++ will take about O(100) LOC and the error comes from the constant-level discrepancies between P4RTC and NPC++ languages (*e.g.*, table definition, 'for loop' syntax). Additionally, we successfully compile CocoSketch [6], ONTAS [53], and AES encryption [15] by only modifying 4.3%-13.0% of P4 code. These modifications mainly involve the architecture-specific extern constructs (*e.g.*, RegisterAction), and similar extern constructs can be easily found in P4RTC.

Table 2: Lines of Code (LOC). 'Modification (Mod.)' denotes the P4 code modifications required to migrate the original project to P4RTC. The required LOCs are comparable to NPC++, in the O(100) range.

Project	Microcode	$P4_{16}$ (Mod.)
Accurate Flow Mon.	2825	547 (N/A)
SpaceSaving [52]	1166	152 (N/A)
CocoSketch [6]	1244	270 (35)
AES Encryption [15]	2883	561 (41)
ONTAS [53]	3029	637 (27)

7.3 Case#3: Using the Performance Model

This case specifically aims to evaluate how the performance model can identify bottlenecks within the memory subsystem and guide optimization. Figure 11 is a sample P4 code we use when opening up the system process. The baseline deployment does not take any optimization. As shown in Table 3, its throughput is less than 40 Mpps. Then, we repeatedly used the performance model to guide the optimization.

Firstly, observing the status of memory banks in the baseline, we find that the utilization gap among the banks is significant. The maximum utilization is 0.99, the minimum is 0.03, and the average is low. This prompts us to adopt table fragments and use the load-aware table deployment strategy discussed in §5. After this optimization (O1), the system's performance increased 7.2 times.

After O1, we further find that load imbalance also existed among the table search engines (TSEs). We use a similar load-aware strategy to rebind the P4 tables to multiple TSEs (O2), balancing the loads of multiple TSEs. This performance optimization increases performance by 1.38 times.

Thirdly, when the bottleneck of TSEs is eliminated, the bottleneck point is once again transferred to the memory subsystem. We find all memory banks' utilization is above 90%. This reveals that the off-chip deployment of the P4 tables limits the system throughput. We offloaded a small number of hotspot table entries to on-chip memory (O3). This further improves performance to 1.45x over O2.

The Accuracy of the performance model. In the above

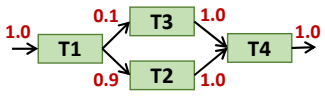


Figure 11: Case 3 P4 program. All off-chip tables.

Table 3: Performance analysis with the performance model.

Opt.	Banks Util. (Min / Max) Avg	TSEs Util. (Min / Max) Avg	Throughput Model (Mpps)	Throughput Hardware (Mpps)
Baseline	0.03 / 0.99 0.09	0 / 0.09 0.04	39.4	38.3
+ O1. Table Redeploy	0.63 / 0.90 0.68	0.03 / 0.63 0.32	283.9	279.4
+ O2. TSE Rebind	0.90 / 0.99 0.95	0.44 / 0.44 0.44	392.7	401.9
+ O3. Table Caching	0.77 / 0.96 0.82	0.63 / 0.63 0.63	569.5	561.0

experiments, the overall error of the performance was less than 3%, which is sufficient for the optimization scenario. We discuss the lessons and current limitations about the accuracy of the performance model in §8.

8 Lessons and Discussion

P4 language design for the run-to-completion architecture. The expressiveness of the RTC architecture is not limited to the externs proposed in this paper. In practice, we also developed some more specialized P4 extensions based on application requirements, such as using `@cache` annotation to directly support the table caching mechanism. Considering openness, the ultimate design of the P4 language for the RTC architecture merits discussion within open communities.

Performance challenges introduced by programmability. The NE8000F1A series comprises high-performance commodity routers designed with fixed functions that are optimized for their functional logic and configurations. Upon introducing programmability, we encountered numerous performance challenges. Some early applications consistently achieved throughputs of less than 100 Gbps. Contrasting with conventional deterministic performance switches, both toolchain developers and users need to invest additional effort to achieve the best performance.

Accuracy boundary of the performance model. Replicating all hardware details in a cycle-accurate manner is a monumental engineering challenge. The modeling granularity directly affects the accuracy and execution efficiency of the performance model. Currently, the performance model is built based on actual needs to maintain satisfactory prediction accuracy in the typical (or known) scenarios (*e.g.*, table deployments). Due to the probabilistic execution of the *codepath*, the current model cannot simulate some detailed behaviors, such as complex locking mechanisms and scenarios where the specific order of packets matters. Our ongoing development is focused on refining the model's traffic generator and chip simulator to support a richer array of scenarios and performance behaviors.

Parallel safety issues in P4RTC. Users must use the `Lock` extern reasonably according to their scenarios and consider potential performance risks. Better-performance concurrency management methods are worthy of future research, especially case-by-case optimizations and trade-offs. Our exploration in Appendix C provides an optimization case.

Future work. We will enrich P4RTC and adapt it to our higher-throughput RTC chips. Support for P4 Runtime is also in the planning process. Besides, we are applying P4RTC to more practical projects, such as traffic generators [54], in-network storage [1], and ML training acceleration [55].

9 Related Work

RMT [11] makes the concept of programmable switches a reality. Nevertheless, RMT is plagued by sub-optimal memory utilization due to the coupling between memory and the MAU stage. dRMT [56] enhances memory utilization and improves execution efficiency by moving table memories out of pipeline stages and into a centralized pool. P4ALL [26] and O4 [57] are extensions of P4 that support elastic switch programming and reduce the voluminosity of P4 code, respectively. Lyra [58] is a cross-platform language for data plane programming. Although these efforts simplify function development and enhance resource utilization, they are not designed to overcome the inherent limitations of P4's programmability within pipeline architectures.

There have been some applications and research around RTC architecture devices. Turboflow [59] employs Netronome NFP-4000 for implementing network flow records generation. Trio [31] is Juniper's programmable chipset using RTC architecture. The language design of P4RTC, along with its experience in compiler optimization and performance modeling, can serve as a reference for Trio. P4TC [28] focuses on applying the P4 language to the Linux tc layer. It is still programmed from a pipeline perspective. Designing new externs to extend the P4 language to the RTC architecture and leveraging the RTC architecture are not within the scope of its design.

10 Conclusion

We proposed P4RTC, a comprehensive summary of our experiences in applying the P4 language to the RTC architecture. P4RTC includes a unified programming model with novel extern constructs, compilers, and a performance model. Case-oriented evaluation demonstrates that P4RTC can enhance P4 programmability and reduce RTC development burdens.

Acknowledgments

We thank our shepherd, Srinivas Narayana, and the anonymous reviewers for their constructive feedback. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204 , the Fundamental Research Funds for the Central Universities, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 121–136. [Online]. Available: https://dl.acm.org/doi/10.1145/3132747.3132764

[2] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "NetLock: Fast, Centralized Lock Management Using Programmable Switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 126–138. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405857

[3] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 44–58. [Online]. Available: https://dl.acm.org/doi/10.1145/3341302.3342085

[4] K. Liu, C. Tian, Q. Wang, H. Zheng, P. Yu, W. Sun, Y. Xu, K. Meng, L. Han, J. Fu, W. Dou, and G. Chen, "Floodgate: taming incast in datacenter networks," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 30–44. [Online]. Available: https://doi.org/10.1145/3485983.3494854

[5] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2016, pp. 1–12.

[6] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: high-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 207–222. [Online]. Available: https://doi.org/10.1145/3452296.3472892

[7] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "{SketchLib}: Enabling Efficient Sketch-based Monitoring on Programmable Switches," 2022, pp. 743–759. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/namkung

[8] ——, "Sketchovsky: Enabling ensembles of sketches on programmable switches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1273–1292. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/namkung

[9] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021, conference Name: IEEE Access.

[10] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–7.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[12] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 164–176. [Online]. Available: https://dl.acm.org/doi/10.1145/3050220.3063772

[13] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 313–323, iSSN: 1092-1648.

[14] "Intel halts Tofino networking chip development - Silicon Valley Business Journal." [Online]. Available: https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html

[15] X. Chen, "Implementing aes encryption on programmable switches via scrambled lookup tables," *ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure (SPIN 2020)*, 2020.

[16] S. Vaucher, N. Yazdani, P. Felber, D. E. Lucani, and V. Schiavoni, "ZipLine: in-network compression at

line speed," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 399–405. [Online]. Available: https://doi.org/10.1145/3386367.3431302

[17] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, and X. Jin, "{NetVRM}: Virtual register memory for programmable networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 155–170.

[18] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 518–532. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230548

[19] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 490–503. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230582

[20] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "bf4: towards bug-free p4 programs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 571–585.

[21] M. A. Noureddine, A. Hsu, M. Caesar, F. A. Zaraket, and W. H. Sanders, "P4aig: Circuit-level verification of p4 programs," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*. IEEE, 2019, pp. 21–22.

[22] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 1–16.

[23] B. Vass, C. Sarkadi, and G. Rétvári, "Programmable packet scheduling with sp-pifo: Theory, algorithms and evaluation," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2022, pp. 1–6.

[24] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 685–699. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/sharma

[25] R. Doenges, M. T. Arashloo, S. Bautista, A. Chang, N. Ni, S. Parkinson, R. Peterson, A. Solko-Breslin, A. Xu, and N. Foster, "Petr4: formal foundations for p4 data planes," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–32, 2021.

[26] M. Hogan, S. Landau-Feibish, M. Tahmasbi Arashloo, J. Rexford, D. Walker, and R. Harrison, "Elastic switch programming with p4all," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 168–174. [Online]. Available: https://doi.org/10.1145/3422604.3425933

[27] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing P4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, ser. EuroP4 '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 19–25. [Online]. Available: https://doi.org/10.1145/3565475.3569078

[28] J. Hadi Salim, D. Chatterjee, V. Nogueira, P. Tammela, T. Osinski, E. Haleplidis, B. Sambasivam, U. Gupta, K. Jain, and S. Sethuramapandian, "Introducing P4TC - A P4 implementation on Linux Kernel using Traffic Control," in *Proceedings of the 6th on European P4 Workshop*, ser. EuroP4 '23. New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 25–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3630047.3630193

[29] P. Community, "p4-dpdk-target," https://github.com/p4lang/p4-dpdk-target, 2023, accessed: 2024-8-30.

[30] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2018, pp. 1–8, iSSN: 2325-5609. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8850752

[31] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, "Using trio: juniper networks' programmable chipset - for emerging in-network applications," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser.

SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 633–648. [Online]. Available: https://dl.acm.org/doi/10.1145/3544216.3544262

[32] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2go: P4 profile-guided optimizations," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 146–152.

[33] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190558

[34] H. Zolfaghari, H. Mustafa, and J. Nurmi, "Run-to-Completion versus Pipelined: The Case of 100 Gbps Packet Parsing," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2021, pp. 1–6, iSSN: 2325-5609.

[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2656877.2656890

[36] "P4~16~ Portable Switch Architecture (PSA)." [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[37] D. D. Robin and J. I. Khan, "P4TE: PISA switch based traffic engineering in fat-tree data center networks," *Computer Networks*, vol. 215, p. 109210, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912862200295X

[38] I. Tofino, "Tofino2," May 2023. [Online]. Available: https://www.intel.com/content/www/cn/zh/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[39] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 313–323, iSSN: 1092-1648.

[40] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang, "{LightGuardian}: A {Full-Visibility}, Lightweight, In-band Telemetry System Using Sketchlets," 2021, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/zhao

[41] "p4c," Aug. 2023, original-date: 2016-04-04T18:12:32Z. [Online]. Available: https://github.com/p4lang/p4c

[42] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403001925

[43] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, Apr. 2001, pp. 1454–1463 vol.3, iSSN: 0743-166X.

[44] e. a. Wei Bai, "Empowering azure storage with RDMA," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 49–67. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/bai

[45] "BEHAVIORAL MODEL (bmv2)," Sep. 2023, original-date: 2015-01-26T21:43:23Z. [Online]. Available: https://github.com/p4lang/behavioral-model

[46] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-Aware Performance Prediction For Virtualized Network Functions," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 270–282. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405868

[47] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance Contracts for Software Network Functions," 2019, pp. 517–530. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/iyer

[48] "SystemC," Aug. 2023, page Version ID: 1172583305. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SystemC&oldid=1172583305

[49] J. Xing, Y. Qiu, K.-F. Hsu, S. Sui, K. Manaa, O. Shabtai, Y. Piasetzky, M. Kadosh, A. Krishnamurthy, T. S. E. Ng, and A. Chen, "Unleashing SmartNIC Packet Processing Performance in P4," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, Sep. 2023, pp. 1028–1042. [Online]. Available: https://dl.acm.org/doi/10.1145/3603269.3604882

[50] "Microarchitecture - an overview | ScienceDirect Topics." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/microarchitecture

[51] S. Panchen, N. McKee, and P. Phaal, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," Internet Engineering Task Force, Request for Comments RFC 3176, Sep. 2001, num Pages: 31. [Online]. Available: https://datatracker.ietf.org/doc/rfc3176

[52] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams," in *Database Theory - ICDT 2005*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Berlin, Heidelberg: Springer, 2005, pp. 398–412.

[53] H. Kim and A. Gupta, "Ontas: Flexible and scalable online network traffic anonymization system," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, 2019, pp. 15–21.

[54] Y. Chen, B. Tian, C. Tian, L. Dai, Y. Zhou, M. Ma, M. Tang, H. Zheng, Z. Yang, G. Chen, D. Cai, and E. Zhai, "Norma: Towards Practical Network Load Testing," 2023, pp. 1733–1749. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/chen-yanqing

[55] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling Distributed Machine Learning with {In-Network} Aggregation," 2021, pp. 785–808. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sapio

[56] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "Drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14. [Online]. Available: https://doi.org/10.1145/3098822.3098823

[57] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing p4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 19–25.

[58] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 435–450. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405879

[59] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190558

[60] "LLVM Assembly Language Reference Manual." [Online]. Available: https://releases.llvm.org/1.1/docs/LangRef.html

[61] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2012, pp. 160–174.

[62] K. Cho, K. Mitsuya, and A. Kato, "Traffic data repository at the wide project," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '00. USA: USENIX Association, 2000, p. 51.

Appendix

Appendices are supporting material that has not been peer reviewed

A. Example of the Queue extern

In the following example, the foreground pipeline uses the Queue extern to send signals of finished flows to background pipelines. The motivation behind this design is this. First, the foreground pipeline offloads the reported tasks to the background pipeline for better logical separation. In this way, the cores running the foreground pipeline do not need more packet processing time when reporting is required. And the background pipeline can implement advanced reporting logic, such as batching and compression. However, it is inefficient for the backend pipeline to blindly scan the flow table to determine whether it needs to be aged. A simple optimization method is to pass the end-of-flow signal from the foreground pipeline to the background pipeline through the Queue to achieve more intelligent and efficient table aging.

```
Queue<FlowInfo> queue(4096);
control foreground_ingress{
  @offchip
  table FlowTable {...}
  ...
  apply{
    if (hdr.tcp.flags & TCP_FIN
              || hdr.tcp.flags & TCP_RST)
    {
        queue.push( { hdr.ipv4.srcAddr,
                      hdr.ipv4.dstAddr,
                      hdr.ports.srcPort,
                      hdr.ports.dstPort} );
    }
    /* Measurement logic here, store flow data
      in the action data of FlowTable */
  }
}
control background_ingress{
  @tbl(FlowTable)
  TableOperation() tbl_op;
  apply{
     /* Get an expired flow record */
     FlowInfo flow;
     queue.pop(flow);
     /* Read flow infomation*/
     bit<8> action_id;
     FlowData action_data;
     tbl_op.entryGet(flow, action_id,
                           action_data);
     ...
     /* Deleta the flow record*/
     tbl_op.entryDel(flow);
     /* Set valid the report header and send to
      the measurement analyzer. */
  }
}
```

B. Example of Microcode Generation

The following example introduces how to convert the `Foreach` extern into the final microcode step by step. Since the specific microcode cannot be made public for confidentiality reasons, we use LLVM assembly language [60] for demonstration, which is very close to our final microcode.

The P4 language snippet we're working with introduces an extern library for table traversal, named `Foreach`. This library also contains methods such as `continue()` and `break()`, which control the traversal flow. The `Foreach` extern is used in the P4 program as follows:

```
action act0(bit<32> value) {}
action act1(bit<32> value0,
            bit<32> value1) {}
table t {
    key = {
        user_meta.key : exact;
    }
    actions = { act0(); act1(); }
    size = 1024;
}

/* Add '@foreach' annotation to indicate this
   action is a method in 'Foreach' procedure */
@foreach
action __for_act0(bit<32> key,
                  bit<32> val) {
    // ...
    continue();
    /* Use 'continue' extern function to
       continue to the next iteration */
}
@foreach
action __for_act1(bit<32> key,
                  bit<32> val0,
                  bit<32> val1) {
    // ...
    break();
    /* Use 'break' extern function to
       break the loop */
}

// Instantiate a 'Foreach' variable.
// Bind this instance to table 't'
// Bind method '__for_act0'
//      to action 'act0' of table 't'
// Bind method '__for_act1'
//      to action 'act1' of table 't'
@tbl("t")
@methods[act0="__for_act0", act1="__for_act1"]
Foreach() foreach_t;
apply {
    // ...
    foreach_t.apply();
}
```

The `Foreach` extern needs to be bound to a table, and each table action of the table requires a separate `foreach` action (*i.e.*, loop body). This is because different actions usually have various types of action data, and we need to match the function parameter list.

Firstly, the P4C compiler translates the P4 language into the corresponding P4 IR. The P4 IR is a tree structure representing the P4 code in a more abstract form, including information about tables, actions, and extern constructs.

```
P4Action name=act0
  parameters: ParameterList
    Parameter name=value
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
P4Action name=act1
  parameters: ParameterList
    Parameter name=value0
      type: Type_Bits size=32 isSigned=0
    Parameter name=value1
      type: Type_Bits size=32 isSigned=0
```

```
  body: BlockStatement
    <... body ...>
P4Table name=t
  properties: TableProperties
    Property name=actions isConstant=0
      value: ActionList
        <... act0 ...>
        <... act1 ...>
    Property name=key isConstant=0
      value: Key
        <... user_meta.key ...>
P4Action name=__for_act0
  annotations: Annotations
    Annotation name=foreach
  parameters: ParameterList
    Parameter name=key
      type: Type_Bits size=32 isSigned=0
    Parameter name=value
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
P4Action name=__for_act1
  annotations: Annotations
    Annotation name=foreach
  parameters: ParameterList
    Parameter name=key
      type: Type_Bits size=32 isSigned=0
    Parameter name=value0
      type: Type_Bits size=32 isSigned=0
    Parameter name=value1
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
Declaration_Instance name=foreach_t
  annotations: Annotations
    Annotation name=tbl text=t
    Annotation name=methods
      NamedExpression name=act0 value=__for_act0
      NamedExpression name=act1 value=__for_act1
  type: Type_Name name=Foreach
```

We can see that the P4 IR uses the elements in P4 as the main content and is still far away from the final microcode. We first translate the P4 IR to Microcode IR. This transformation involves converting the high-level P4 constructs into lower-level ones that are more suitable for generating microcode instructions. For example, we must build a loop structure, conditions for entering or exiting the loop, and distinguishing the logic of calling different table action's loop bodies (*i.e.*, `SwitchStatement`). Here is the representation of the Microcode IR of the P4 code:

```
BlockStatement
  LabelStmt
    label: Label name=entry
  AssignmentStatement
    left: PathExpression name=i
    right: Constant value=0
  Branch
    target: Label name=cond
  LabelStmt
    label: Label name=cond
  IfStatement
    condition: Lss
      type: Type_Boolean
      left: PathExpression name=i
      right: Constant value=10
    Branch
      target: Label name=body
    Branch
      target: Label name=exit
  LabelStmt
    label: Label name=body
  EntryGet
    table: PathExpression name=t
    key: PathExpression name=idx
    act: PathExpression name=act
    data: PathExpression name=data
  SwitchStatement
    expression: PathExpression name=act
    SwitchCase
      label: Constant value=0
      statement: BlockStatement
        <... body ...>
    SwitchCase
      label: Constant value=1
      statement: BlockStatement
        <... body ...>
  AssignmentStatement
    left: PathExpression name=i
    right: Add
      left: PathExpression name=i
      right: Constant value=1
  Branch
    target: Label name=cond
  LabelStmt
    label: Label name=exit
```

Finally, the Microcode IR is transformed into microcode instructions. After the conversion from P4 IR to Microcode IR, the conversion from Microcode IR to microcode instructions is a straightforward parallel mapping.

```
%union.Data = type { %struct.Act0_Data }
%struct.Act0_Data = type { i32 }
%struct.Act1_Data = type { i32, i32 }

define void @foreach_t() {
     %act_ptr = alloca i8
     %data_ptr = alloca %union.Data
     %idx_ptr = alloca i32
     br label %enter

enter:
     store i32 0, i32* %idx_ptr
     br label %cond

cond:
     %cond_idx = load i32, i32* %idx_ptr
     %cond_result = icmp slt i32 %cond_idx, 100
     br i1 %cond_result, label %body,
           label %exit

body:
     %body_idx = load i32, i32* %idx_ptr
     %body_entry_get_result = call i1
                 @entry_get(i32 %body_idx,
                            i8* %act_ptr,
                       %union.Data* %data_ptr)
     br i1 %body_entry_get_result,
               label %body_switch,
               label %next

body_switch:
     %body_act = load i8, i8* %act_ptr
     switch i8 %body_act, label %next [
          i8 0, label %body_act0
          i8 1, label %body_act1
     ]

body_act0:
     %act0_idx = load i32, i32* %idx_ptr
     %act0_data = bitcast %union.Data*
                  %data_ptr to %struct.Act0_Data*
     %act0_result = call i1
                 @foreach_act0(i32 %act0_idx,
                               %struct.Act0_Data*
                               %act0_data)
     br label %next

```

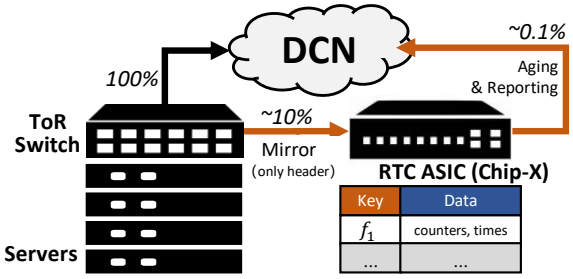


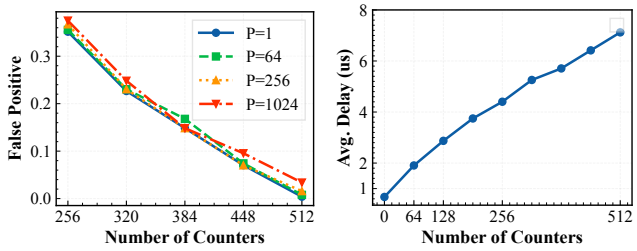
Figure 12: Testbed for Accurate Per-flow Monitoring

```
body_act1:
     %act1_idx = load i32, i32* %idx_ptr
     %act1_data = bitcast %union.Data*
                  %data_ptr to %struct.Act1_Data*
     %act1_result = call i1
               @foreach_act1(i32 %act1_idx,
                   %struct.Act1_Data*
                   %act1_data)
     br label %next

next:
     %next_idx = load i32, i32* %idx_ptr
     %next_new_idx = add i32 %next_idx, 1
     store i32 %next_new_idx, i32* %idx_ptr
     br label %cond

exit:
     ret void
}

declare i1 @entry_get(i32, i8*,%union.Data*)
declare i1 @foreach_act0(i32,%struct.Act0_Data*)
declare i1 @foreach_act1(i32,%struct.Act1_Data*)
```

C. Profiling Complex Algorithms

Algorithms that involve complex memory operations can now be implemented with the help of P4RTC. A typical example is SpaceSaving (SS) [61], which is utilized to detect the top-k flows. SS maintains O(k) slots in memory, where each slot contains a flow key and a counter to track large flows. Upon receiving a packet from flow i, the algorithm employs a loop to verify whether the flow is stored within a slot. If the flow is found, the algorithm increments the corresponding counter. Otherwise, it locates the slot with the minimum counter value and replaces the existing flow with flow i. These kinds of operations cannot be implemented in the pipeline architecture. With P4RTC, we can easily use `Foreach` extern to implement an SS algorithm.

However, parallel safety issues may occur during the updating process. Using locks can lead to significant performance issues as they negate the performance benefits of parallel packet processing. Therefore, we are curious whether SS can be applied directly without the locks. To deeply reveal the impact of the parallel safety issues on algorithm accuracy, we leverage our performance model (in §6) to help us analyze this process. We embed a customized execution environment in the performance model to actually run the actual instructions in the codepath, thus profiling the functional performance in a multi-threaded environment. We input a



(a) SS Accuracy Profiling (b) SS Latency Profiling

Figure 13: Profiling with the performance model

real-world packet trace collected by the WIDE Project in 2020 [62], which contains 10M packets. Figure 13a shows the false negative of the SS algorithm across numbers of cores at 1, 64, 256, and 1024. We detect top-256 large flows ($k = 256$) and the x-axis represents the number of slots allocated for SS. We surprisingly find that we can optimistically neglect the locks. Even when utilizing 1024 cores (parallelism $P = 1024$), the algorithm maintains its accuracy without noticeable degradation. This is primarily due to the stability of the large flow candidates in memory as the algorithm progresses. The accumulation of these atomic counters is parallel safe. The only potential parallel safety concerns arise during the eviction of the smallest flow. Fortunately, this operation does not impact the other slots that contain larger flows, thus the impact of parallel safety issues is limited.

The performance model can also help us balance parameter k (*i.e.*, the number of large flows that need to be identified) and performance overheads. Suppose for each k, we allocate $2k$ slots in memory to obtain a satisfactory recall ratio. Figure 13b shows the variation of average packet latency with the change of allocated slots. Developers can choose the most appropriate parameters k based on their tolerance for delays and the top-k flow requirements.

Table 4: Comparison between P4RTC with regular P4

Language	P4	P4RTC
Target	Pipeline	Run-to-completion
Architecture Model	Pipeline Only	Pipeline and Manycore RTC
Table Management	Control plane Only	Control plane and Data plane
Memory Layout	Isolated memory & Limited access	Shared on-chip/ off-chip Memory
Multi-Program Programming	Physical Pipeline Level	Logical pipeline level for each thread with different roles
Thread Safety	Safe	Unsafe
Performance Guarantee	Deterministic Performance	Varying Performance