P4runpro: Enabling Runtime Programmability for RMT Programmable Switches

Yifan Yang[†], Lin He[†♠], Jiasheng Zhou[◇] Xiaoyi Shi[†], Jiamin Cao[△], Ying Liu[†♠]
[†]Tsinghua University, [♠]Zhongguancun Laboratory, [◇]Fuzhou University, [△]Alibaba Cloud

ABSTRACT

Programmable switches have revolutionized network operations by enabling the flexible customization of packet processing logic using language like P4. However, changing the programs running on the switch requires disturbing traffic and suspending other unrelated programs. In this paper, we present `P4runpro`, enabling runtime data plane updates with dynamic resource allocation. The `P4runpro` data plane abstracts hardware resources and defines dynamically reconfigurable atomic operations that form packet processing logic. `P4runpro` provides runtime programming interfaces called `P4runpro` primitives for the operator to write high-level programs. We have designed the `P4runpro` compiler to automatically and consistently link the `P4runpro` programs to the running data plane. We implement our prototype on a Tofino switch. We implement 15 example runtime programs using `P4runpro` to demonstrate its generality and expressiveness. Our evaluation results show that compared to the state-of-the-art, `P4runpro` can respond within hundreds of milliseconds, achieve an average of 60% to 80% dynamic resource utilization, concurrently run ≈0.6K to ≈2.8K programs, and introduce lower overhead. Our case studies illustrate the benefit of runtime programming and prove the same functionality between `P4runpro` and conventional P4 programs.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**.

KEYWORDS

Runtime programmable switches, P4, RMT

ACM Reference Format:
Yifan Yang, Lin He, Jiasheng Zhou, Xiaoyi Shi, Jiamin Cao, and Ying Liu. 2024. P4runpro: Enabling Runtime Programmability for RMT Programmable Switches. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3651890.3672230

1 INTRODUCTION

The utilization of programmable switches empowers operators to flexibly customize packet processing logic using domain-specific languages like P4 [4]. With both high performance and programmability, programmable switches facilitate the offloading of numerous network functions including in-network computing [35, 40, 53], measurement [41, 43, 59], load balancing [45, 46, 60], security [30, 36, 56, 57], congestion control [11, 18, 42], and more [15, 16] to the switch data plane [29], giving rise to what are known as switch programs.

Despite the substantial advantages gained from utilizing programmable switches in the network community, a new formidable challenge arises due to the increasing number of offloaded switch programs. Current programmable ASICs can only specify data plane programs at compile-time and are unable to update the switch data plane in a runtime manner. When a running program needs to be changed, the operator has to reprovision the switch, causing disruptions to traffic and suspending unrelated switch programs. This limitation stems from P4's limited level of abstraction, *i.e.*, it only defines a single data plane context [28]. As a result, all switch programs defined by P4 are bound to and share the same hardware resources [22, 37].

Several endeavors have been dedicated to overcoming the challenge by enabling runtime reconfiguration or programmability of switch programs, broadly classified into three categories. The first category focuses on architectural extensions [23, 48, 54, 55] of designing or seeking more flexible architectures. However, these solutions may not be compatible with the widely used Reconfigurable Match-Action Table (RMT) switches [14] in academia and industry [38, 46]. The second category involves virtualizing the data plane for runtime switch program reconfiguration [28, 61]. Unfortunately, these efforts introduce significant resource overhead, making them impractical for implementation on hardware targets [64]. The third category aims to achieve dynamic resources on RMT hardware, including dynamic memory [66], dynamic combinations of task components [63, 65], dynamic memory operations [22], and dynamic program call for multi-tenant [37]. Nonetheless, these approaches are either limited to specific applications or impose substantial overhead on the end host, lacking generality.

To bridge the gap, we propose `P4runpro`, which enables runtime programmability for RMT ASICs. This runtime programmability allows for data plane updates without the need for switch reprovisioning, ensuring no disruptions to ongoing traffic and programs. Compared to the state-of-the-art, `P4runpro` can be generalized for heterogeneous switch programs and has a lower overhead. Our core concept is to *decouple the implementation of switch programs from hardware resources.* We achieve this by identifying and abstracting a set of atomic operations from a variety of heterogeneous switch programs. These atomic operations are pre-installed at compile-time and can then be flexibly combined into various target programs at runtime.

Realizing this idea faces three challenges. First, RMT ASICs are originally designed to support a single P4 context, using all shared resources on the chip. As a result, implementing multiple independent programs with adequate resource isolation is extremely resource-consuming [64]. Second, P4runpro relies on the in-switch very long instruction word (VLIW) to implement atomic operations. However, RMT ASICs have very limited VLIW resources, which constrains the capacity of atomic operations. Third, given the limited resources of ASICs and the unpredictable resource demands of each program, designing an efficient resource allocation scheme to dynamically support massive programs is a challenge.

In response, P4runpro proposes the following three designs:

- **P4runpro data plane (§4.1)**: to support multiple isolated programs simultaneously, we abstract the execution logic of heterogeneous programs to general programming units, *i.e.*, runtime programming blocks (RPBs). This allows switch programs to become the execution of multiple rounds of atomic operations in RPBs. The same hardware resources are reused by various programs, thus reducing resource consumption.
- **P4runpro primitive (§4.2)**: to overcome the challenge of limited operation capacity, we design P4runpro primitives and pseudo primitives as runtime programming interfaces.
- **P4runpro compiler (§4.3)**: to dynamically and automatically link the runtime program to the data plane, we design the P4runpro compiler which can translate the input programs into entries and consistently update them to the data plane. The key of the compiler is an efficient resource allocation scheme.

We implement our prototype on an Intel Tofino [32] switch (§5). Our evaluation results (§6) demonstrate that P4runpro: (1) is general enough to express 15 distinct P4 programs from various fields, (2) enables rapid updates, with a response time of just a few hundred milliseconds, (3) achieves an average resource utilization of 60% to 80%, (4) can concurrently run ≈0.6K to ≈2.8K programs, according to their complexity, and (5) introduces a 7% less reduction in throughput loss than prior work while imposing no extra overhead on end hosts. Furthermore, through case studies based on real traffic scenarios, we show that P4runpro has no impact on actual traffic and offers functionality equivalent to that of standalone P4 programs. Our prototype implementation is available at [6].

2 BACKGROUND AND RELATED WORK

This section introduces the background and related work to elucidate our motivation for enabling runtime programmability for RMT programmable switches.

2.1 Background

Conventional P4 Workflow. The conventional P4 workflow on RMT ASICs involves two phases: compile-time and runtime. During compile-time, operators write their P4 programs and then utilize the P4 toolchain (*e.g.*, P4C [2]) to translate the P4 program into a binary image. Once the switch is initiated and enters the runtime phase by provisioning the binary image to the data plane, operators can employ southbound APIs to configure table entries, with predefined match keys and actions from compile-time. Unfortunately, a gap exists in enabling dynamic packet processing logic for RMT ASICs. The components of the switch program, including conditional branches, tables, and stateful memory, are only programmable at compile-time, solidifying into fixed functions at runtime [55]. When operators wish to make changes, a new binary image is required to reprovision the data plane and the process pauses all switch functions, causing disruptions in traffic [22, 63]. In comparison to operating systems on CPUs that offer runtime programmability for high-level languages (*e.g.*, C) and abstract them from the hardware (*e.g.*, Linux uses pids to distinguish different processes and dynamically allocates CPU cores and memory to each process), P4 and its toolchain do not provide resource isolation for independent switch programs. All logically independent switch programs share resources and must be physically started or stopped simultaneously.

Runtime Programmability for RMT Switches. The limitations of the conventional P4 workflow serve as the driving force behind our motivation to introduce hardware resource isolation and dynamic program execution, *i.e.*, runtime programmability, to widely-used RMT switches. RMT switches have been utilized and deployed in academia and industry for several years [38, 46], proving their utility and popularity. Enabling runtime programmability for RMT switches will make networks more responsive, robust, and flexible. Since the changes required for the network are usually minor, runtime programmability enables just-in-time optimizations that the network can adjust without downtime [55]. When the network fails or its performance decreases, the operator can deploy measurement and attack detection tasks in a timely manner [63]. Additionally, In a multi-tenant scenario, runtime programmability facilitates the offloading of network function virtualization from servers to switches, making the switch cloud-native [22, 37].

2.2 Related Work

Hardware Solutions. Some researchers pursue runtime programmability by architectural extensions [23, 48, 54, 55]. Menshen [54] extends the RMT pipeline to achieve isolation and reconfiguration of packet-processing modules. The In-situ Programmable Switch Architecture (IPSA [23]) is a newly designed switch architecture with the ability of runtime programmability. However, these two approaches are only prototyped on FPGA platforms [25, 67], lacking real ASIC implementations and evaluation. FlexCore [48, 55] utilizes the disaggregated Reconfigurable Match-Action Table (dRMT [19]) architecture, enabling runtime programmability through the indirect and partial reconfiguration of the switch data plane. Still, FlexCore cannot be ported to popular RMT switch ASICs due to architectural constraints. In addition, similar to CPU, dRMT's run-to-completion manner causes performance degradation when using additional cycles to achieve partial reconfiguration [37]. Instead, RMT switches guarantee the line rate of packet processing due to their pipelined architecture.

Virtualization. Some efforts focus on virtualizing the data plane [28, 61, 64]. Hyper4 [28] and HyperV [61] virtualize the data plane tables and support reconfiguring switch programs at runtime. However, these two approaches entail such significant overhead that they are difficult to implement on hardware targets, demonstrated in P4Visor [64]. P4Visor virtualizes the data plane at the compile-time by combining multiple P4 programs into one. It is orthogonal to our work since we focus on the runtime.

Dynamic Resources. The closest efforts to our work are enabling dynamic resources on RMT switches [22, 37, 63, 65, 66].

NetVRM [66] is a dynamic memory management system. In NetVRM, the data plane applications' memory can be dynamically extended and allocated based on a utility function. NerVRM only supports dynamic memory of fixed applications which are predefined at compile-time.

Newton [65] and FlyMon [63] target the fine-grained task reconfiguration in specific domains where the program logic is similar. Specifically, Newton enables intent-driven traffic monitoring queries by breaking down queries into four sequential processing steps.

FlyMon facilitates real-time task reconfiguration of network measurements by dividing network measurement tasks into flow keys and flow attributes. These two approaches are limited to supporting only a fixed set of similar switch programs, lacking programmability.

ActiveRMT [22] allows the dynamic allocation of active programs comprising active instructions. Still, ActiveRMT lacks the generality to implement various switch tasks since the instruction set is limited to memory operations. In addition, ActiveRMT introduces significant overhead to the end host and is only applicable to capsule-based active networks. It mandates that all coming packets are attached with an additional active header describing the program, resulting in throughput overhead. Header attaching and other resource-consuming operations, such as memory address computation, translation, and storage, are required to be performed at end hosts.

SwitchVM [37] is a concurrent work that enables dynamic program call and execution for multi-tenant. Similar to P4runpro, SwitchVM decouples switch programs to multiple atomic operations and dynamically configures their execution. SwitchVM carefully designs its data plane program execution unit, enabling multi-instruction-multi-data execution that P4runpro cannot support. However, this design occupies two pipeline stages for a single execution unit, which results in nearly half of the stateful memory wasted and a shorter program dependency. P4runpro uses only one stage for each execution unit and has 5.5× execution units than SwitchVM. In addition, SwitchVM attaches additional data to the header and brings additional overhead on the end host and throughput like ActiveRMT.

Summary. Existing approaches either cannot be adapted to commercialized RMT hardware, lack general programmability, or bring additional overhead on the end host and throughput. P4runpro targets general runtime programming on RMT ASICs. P4runpro enables dynamic and isolated resources, boasts general programming interfaces, and makes no assumptions for coming packets.

3 P4RUNPRO OVERVIEW

Before delving into the design details of P4runpro, we first provide an overview. This section first illustrates the architecture of P4runpro and then demonstrates the P4runpro program and workflow by showcasing an example.

3.1 Architecture

The RMT ASIC comprises an ingress pipeline and an egress pipeline, with a traffic manager in between to maintain queues and execute forwarding logic. The two pipelines consist of a series of stages with match-action tables and stateful memory. The packet header vector (PHV) contains stateless data including headers, intrinsic metadata, and customized metadata. The control plane provides a set of raw APIs for operators to configure table entries and gather data plane states. To decouple switch program implementation from hardware resources, we abstract three "registers" in the PHV and virtualize the memory. We pre-install a set of general operations interacting with the header, registers, and memory at compile-time. Then the control plane establishes the execution logic for these operations at runtime, thereby constituting the switch programs. Figure 1 illustrates the architecture of P4runpro, comprising data plane blocks, a primitive set, and a control plane controller.

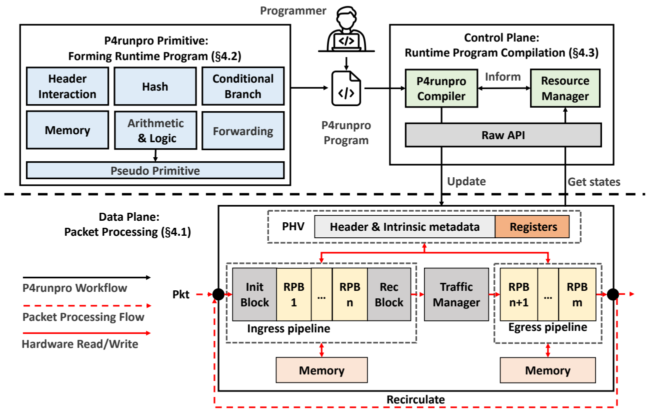


Figure 1: P4runpro architecture.

Data Plane. The data plane of P4runpro, utilized for packet processing, consists of the following key components:

- *Register*: to reuse the PHV in distinct programs, we arrange three "registers" in it, serving as storage for stateless variables during program execution.
- *Initialization Block*: we place it at the first stage of the ingress pipeline to filter traffic based on parsing results.
- *Recirculation Block*: we place it at the last stage of the ingress pipeline to recirculate packets when a program cannot be completed in one circle.
- *RPB*: RPBs can execute the atomic operations that constitute switch programs. RPBs occupy all stages in both the ingress and egress pipelines, excluding those occupied by the other two blocks.
- *Memory*: the memory associated with each stage can be accessed by the corresponding RPB, and a mask-based address translation mechanism is employed to enable dynamic memory.

When a packet arrives, it traverses these blocks sequentially. In the case that the packet is flagged for recirculation, it is directed from the egress port to a designated ingress port, re-entering the pipeline. Otherwise, it is forwarded to the external network.

Primitives. To provide runtime programming interfaces, a set of primitives is supplied, divided into six types. Each primitive, along with its arguments, specifies a single atomic operation in the data plane. A pseudo primitive is expressed by multiple primitives. The composition of primitives outlines the complete packet processing logic.

```
1  @ mem1 1024
2  program cache(
3      /*filtering traffic*/
4      <hdr.udp.dst_port, 7777, 0xffff>) {
5      EXTRACT(hdr.nc.op, har); //get opcode
6      EXTRACT(hdr.nc.key1, sar); //get key[0:31]
7      EXTRACT(hdr.nc.key2, mar); //get key[32:63]
8      BRANCH:
9      /*cache hit and cache read*/
10     case(<har, 1, 0xffffffff>,
11         <sar, 0x00000000, 0xffffffff>,
12         <mar, 0x00008888, 0xffffffff>
13     ) {
14         RETURN; //return to client
15         LOADI(mar, 512); //load address
16         MEMREAD(mem1);  //read cache
17         MODIFY(hdr.nc.value, sar);
18     }
19     /*cache hit and cache write*/
20     case(<har, 2, 0xffffffff>,
21         <sar, 0x00000000, 0xffffffff>,
22         <mar, 0x00008888, 0xffffffff>
23     ) {
24         DROP; //drop the packet
25         LOADI(mar, 512); //load address
26         EXTRACT(hdr.nc.val, sar); //get value
27         MEMWRITE(mem1); //write cache
28     };
29     FORWARD(32); //cache miss
30 }
```

Figure 2: `P4runpro` program for in-network cache.

Control Plane. The `P4runpro` control plane is responsible for program compilation and resource management, comprising two essential components:

- `P4runpro` *Compiler*: it parses the input `P4runpro` program, calculates the resource allocation scheme, and consistently updates the program to the data plane.
- *Resource Manager*: the resource manager maintains dynamic resource usage and monitors memory values.

When the `P4runpro` compiler requires resource allocation for a new program, the resource manager supplies data plane resource usage information to the compiler for allocation computing. Upon allocating or revoking a program, the resource manager updates its data accordingly. After provisioning the `P4runpro` data plane to the switch, the `P4runpro` control plane can be initialized. Then the `P4runpro` programs can be linked to or revoked from the data plane by invoking the `P4runpro` compiler. During the life cycle of a switch program, its states can be monitored by the resource manager.

3.2 Example and Workflow

In this section, we begin by presenting an example of the `P4runpro` program, followed by an outline of its workflow.

Example. We use the example of in-network cache, a component of NetCache [35]. An in-network cache stores values transmitted from a server to a client in an on-path programmable switch, thereby reducing latency and offloading server function. A basic in-network cache involves two types of packets: cache read and cache write. The client-sent cache read packet includes the key, while the server-sent cache write packet includes the key-value pair. The switch performs an initial check for a cache hit on the key. If a cache hit is detected, the switch uses the cache opcode to identify the packet type. Otherwise, the switch forwards the packet to the server. In the case of a cache read packet, the switch embeds the cache value into the packet header and returns it to the client. For a cache write packet, the switch stores the cache value in memory and drops the packet. The switch control plane maintains table entries that store the mappings of keys to cache hits and memory bucket addresses.

Figure 2 depicts the implementation of the in-network cache using the `P4runpro` program. For simplicity, the cache comprises only one key-value pair with a 64-bit key (`0x8888`) and a 32-bit value. The virtual memory bucket address storing the value is `512`. To begin, the `P4runpro` program utilizes the symbol `@` to instruct the compiler about the request for a memory block labeled `mem1` with a size of 1024×32 bits. Subsequently, the `P4runpro` program declares a switch program named `cache` using the `program` keyword. It employs a ternary match tuple (line 4) to specify that the program `cache` is executed for UDP packets with the destination port `7777`. Following this declaration, the program employs `P4runpro` primitives (depicted in orange) along with their arguments to define the packet processing logic. Firstly, the `EXTRACT` primitives (line 5 to line 7) extract the opcode and two 32-bit parts of the cache key from the header, storing them in the registers `har`, `sar`, and `mar`, respectively. Then, the `BRANCH` primitive (line 8) generates a conditional branch by checking the values in these registers for a cache hit and packet type based on the rule tuples (line 10 to line 12) set within the `case` block. If a cache hit occurs and the packet is identified as a cache read packet, the program enters the first `case` block (line 13 to line 18), where `RETURN` instructs the data plane to send the packet back later, `LOADI` loads the memory address into the `mar`, `MEMREAD` retrieves the cache value into the `sar` according to the address stored in the `mar`, and `MODIFY` writes the cache value to the header. Similarly, the primitives `DROP`, `LOADI`, `EXTRACT`, and `MEMWRITE` are employed to handle cache write packets in another branch (line 24 to line 27). In case none of the cases match the values in the three registers, signifying a cache miss, the `FORWARD` primitive instructs the packet to be forwarded to the server behind egress port `32`.

Workflow. The black arrow depicted in Figure 1 illustrates the `P4runpro` workflow. In contrast to conventional P4 workflow, the `P4runpro` data plane remains fixed. The operator only needs to provision the switch once. Subsequently, when an operator intends to link a program to the running switch, he or she simply needs to write the straightforward `P4runpro` program as shown in Figure 2 and invoke the `P4runpro` compiler through `P4runpro` control plane APIs. The compiler then analyzes the program and automatically allocates switch resources accordingly. If the allocation is successful, the `P4runpro` compiler generates corresponding table entry configurations and consistently updates them to the data plane without disrupting traffic or any other program. Except for data plane memory access and table management, the basic control plane functions are the same in `P4runpro` and conventional P4 workflow. The resource manager can implement the data plane memory access by performing an address translation from a virtual address to a physical address. `P4runpro` programs use a `BRANCH` primitive and `case` blocks to express match-action logic instead of a table in the conventional P4 workflow. The incremental update of a running program implements the entry update in conventional P4 workflow.

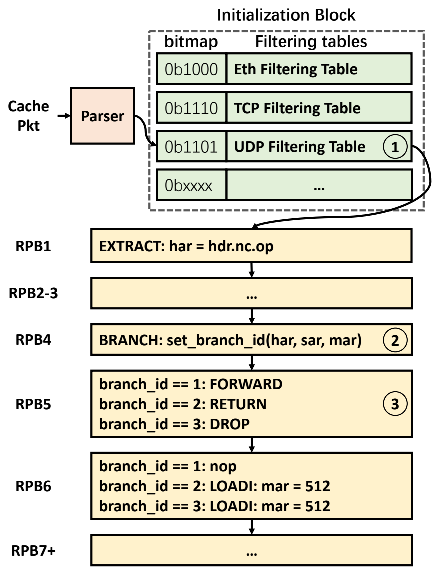


Figure 3: Packet processing for program cache.

4 DESIGN DETAILS

4.1 P4runpro Data Plane

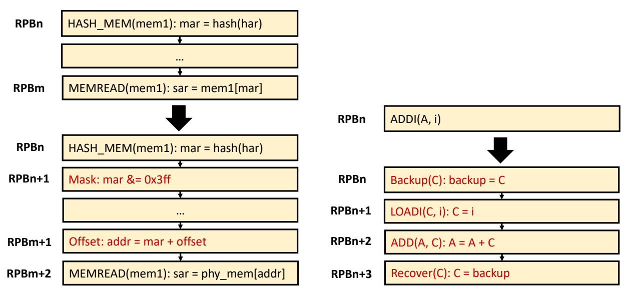
The P4runpro data plane, used for packet processing, is divided into three parts. First, the initialization block identifies the input packet and determines the running program to be executed for it (§4.1.1). Then, the RPBs execute the program logic sequentially, processing the packet and modifying data plane states (§4.1.2). Finally, the packet is either forwarded externally or recirculated, depending on whether the program is completed (§4.1.3). Figure 3 illustrates this process of the program cache.

4.1.1 Flow Filtering. P4runpro enables program isolation at both flow and port granularity. Before entering the pipeline, the packet is parsed following the parsing state machine [13]. We maintain a parsing state bitmap in the PHV and update the corresponding bit when the parsing state machine arrives at a new state. Specifically, each bit in the bitmap represents the existence of a particular header. When the data plane parses a particular header, the corresponding bit representing that header is set to 1. For example, a 4-bit bitmap can represent the existence of the L2 to L4 protocol sequence of Ethernet, IPv4, TCP, and UDP. The bitmap will be 0b1000 if the packet is an L2 packet and 0b1101 if it is a UDP packet. After header parsing, the initialization block filters the packet according to the bitmap. Each parsing path triggers a corresponding filtering table, such that a cache packet, for instance, triggers a UDP filtering table (①). The only action performed by these filtering tables is assigning a unique program ID for the packet according to the filtering rules (line 4 in Figure 2). Subsequent blocks then distinguish and isolate different programs based on the program ID. Program isolation can occur at both flow and port granularity, precisely matching the 5-tuple and ingress port, and can also be coarser, such as matching an address range with a mask.

4.1.2 Program Execution. We decouple the implementation of switch programs from hardware resources to facilitate runtime programmability for RMT switches. We abstract the hardware resources of stateless PHV, stateful memory, and computing units, respectively. Inspired by the register allocation process during the compilation of high-level languages into assembly code, we organize three registers—namely, hash register (har), SALU register (sar), and memory address register (mar)—within the PHV to efficiently reuse PHV storage. These registers serve as distinct variables in various programs. We set the register number as three based on a comprehensive consideration of programming flexibility and resource constraints. Specifically, setting too many registers rapidly increases the resource consumption that stores the atomic operations. For example, enabling operation ADD expressed by $a = a + b$ for 3 registers needs $C_3^1 C_2^1 = 6$ distinct atomic operations while $C_4^1 C_3^1 = 12$ for 4. In contrast, two registers cannot achieve some atomic operations, reducing the flexibility (§4.2). To virtualize stateful memory, we employ a mask-based address translation mechanism before memory operations. In terms of the computing units including PHV ALU, stateful ALU (SALU), and hash units, we establish a set of general atomic operations for invoking them to interact with headers and manipulate registers and memory. These atomic operations are derived from program statements, and their sequential execution forms an entire program. To enable runtime reconfiguration and allocation of these operations, the execution process is implemented by a match-action table in RPB.

RPB. We employ three control flags, *i.e.*, table keys, in the PHV to signify which operation should be executed in the RPB. The first flag is the global program ID, set by the initialization block (①), indicating the current program. The second flag is the program-local branch ID, which is set by the preceding RPB when executing the primitive BRANCH. During the BRANCH execution, the RPB also checks the three registers. A successful match of rules in a specific case block will transition the program into a new branch by setting a new branch ID (②). The primitive from different branches can be executed in one RPB, distinguished by the branch ID (③). The third flag is the packet-local recirculation ID, indicating how many times the packet has been recirculated. This ID is set by the recirculation block. Ingress RPBs can perform operations related to packet forwarding (*e.g.*, return, drop, forward, and report), while egress RPBs cannot. This is because forwarding behaviors are executed by the traffic manager before the packets proceed to the egress pipeline.

Address translation. The physical memory block and hash output width in the data plane are fixed during runtime. Thus we require an address translation mechanism to convert the virtual memory address set by the control plane or calculated by the data plane to a physical memory address. While NetVRM [66] virtualizes memory by treating buckets at the same physical memory address per stage as different buckets of virtual memory, this cross-stage design is not suitable for the per-stage manner of RPB. FlyMon [63] proposes both a shift-based and a TCAM-based address translation mechanism. However, these two mechanisms demand significant VLIW and stage or VLIW and TCAM resources. Consequently, we propose a mask-based address translation mechanism to achieve memory virtualization, consisting of two steps. In the first step, for the address calculated by the hardware hash unit, we apply a mask to adjust the hash output width to match the virtual address declared in the P4runpro program. For example, in Figure 4(a), if the hash output width is 16, the mask will be 0x3ff for mem1 with



(a) Mask-based address translation.   (b) Pseudo primitive translation.

Figure 4: Address translation and pseudo primitive translation.

a size of 1024. Then, the adjusted address is added to an offset on the physical memory to become the physical address. In the second step, for the address set by the control plane (*e.g.*, when the program cache uses LOADI to set the mar), we entrust the programmer with the responsibility of ensuring valid memory access by guaranteeing that the mar will not overflow in program logic. Thus, these addresses do not need the mask step. Both the mask step and the offset step can be executed by one table action in RPB, resulting in significant savings in VLIW resources. The mask step is executed right after the hash operation for memory to ensure that any overflowed hash output is invisible to later primitives. The offset step is executed just before the memory operations, and the result is stored in an additional PHV field, serving as the physical memory address to avoid altering the value of mar. This mechanism supports virtual memory sizes that are powers of 2.

Similar to FlyMon, we utilize the capability of SALU to execute a conditional comparison before memory access, enabling support for additional memory operations. We maintain a SALU flag to indicate which one of the two memory operations will be executed by the SALU. This flag is set concurrently with the offset step.

4.1.3 Recirculation. Ideally, a P4runpro program could be of arbitrary length, but in hardware, the constrained number of pipeline stages imposes limitations on the number of RPB executions. To overcome this, we leverage the recirculation function of the hardware to allow for longer programs. The RPB distinguishes the same packet across different recirculation iterations based on the packet-local recirculation ID. During packet processing, all stateless data, including registers, flags, and addresses, is attached to the packet header for the next cycle of program execution. Note that this additional header remains invisible to the external network. Recirculation can also be replaced by multiple switches deployed on the same path, depending on the volume of traffic and the network topology.

4.2 P4runpro Primitive

To provide runtime programming interfaces, we have crafted a set of P4runpro primitives. These primitives, along with their respective arguments, correspond to the atomic operations in RPBs. However, due to the highly constrained hardware VLIW resources, the operation capacity is limited, preventing us from pre-installing all the atomic operations the hardware can support. Fortunately, we have observed that many switch programs can be implemented in a more restricted operation set, not requiring entire switch functions. This observation is also illustrated in the concurrent work [37]. Based on this key observation, we break down the switch programs into six types of atomic operations: header interaction, hash, conditional branch, memory, arithmetic & logic, and forwarding. The header interaction operations copy the header or intrinsic metadata fields to registers or modify them using the registers' value. The hash operations take har or the 5-tuple as input and store the output in har or mar. The conditional branch operation corresponds to the primitive BRANCH. The memory operations work on the physical memory bucket according to the virtual address stored in mar. The operands of memory operations include the bucket value and sar, and the result is stored back in sar. The operands for arithmetic & logic operations can be any of the three registers. The forwarding operations modify the intrinsic metadata for the traffic manager, specifying the packet's forwarding behavior.

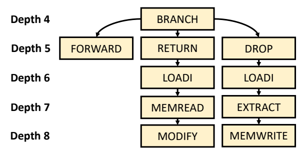
Despite this, the operation number is still limited due to the argument combination. For example, if we implement all operations of the header interaction primitive EXTACT that copies a header field to a register, we need to enumerate all the combinations of header fields and registers. Thus, we carefully design the six types of primitives and introduce the concept of pseudo primitive to ensure generality while utilizing VLIW resources.

Design Principle. The design of P4runpro primitives and pseudo primitives is guided by three principles. First, each RPB needs to support all the primitives to keep equivalence. Second, the primitives and pseudo primitives should support all the arithmetic & logic operations in P4 for enough generality. Third, to leave enough space for header interaction primitives, we compress the remaining primitives as much as possible. For the sake of space, we enumerate all primitives and pseudo primitives with expressions and their argument types in Table 3 and Table 4 in Appendix §A.1.

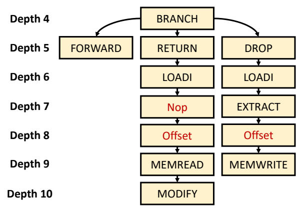
Pseudo Primitive. Similar to pseudo instructions in assembly language [8], the pseudo primitive aligns with the primitive during programming but splits into one or more primitives for execution in hardware. For example, the pseudo primitive ADDI can be translated into sequentially executed primitives LOADI and ADD. We also utilize the addition overflow behavior of the hardware ALU to achieve SUB and SUBI. Specifically, we express $a - b$ as $a + C(b)$ where $C(b)$ is the two's complement of b. We provide all translations of pseudo primitives in Figure 14 in Appendix §A.2. Certain translations of pseudo primitives require an extra supportive register, *i.e.*, another register not used in the pseudo primitive arguments. To maintain the original value of the supportive register, we back it up before and recover it after primitive executions. This process is elucidated in Figure 4(b). To optimize this process, we adopt the concept of register lifetime from register allocation literature [39, 47], eliminating the need to back up the supportive register once it is no longer "live".

4.3 P4runpro Compiler

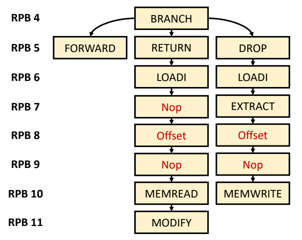
Linking the P4runpro programs to the data plane is a complex process, requiring detailed knowledge of data plane implementation. In contrast to SwitchVM [37] which links and allocates programs in a manual and simple way, we use the P4runpro compiler to dynamically and automatically link and allocate P4runpro programs. The P4runpro compiler is responsible for parsing the input program



(a) AST generated from program synthesis.



(b) AST after primitive translation.



(c) Result of program allocation when the free memory in RPB9 is not enough.

Figure 5: Compilation process of program cache.

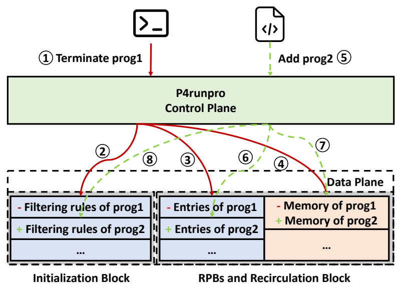


Figure 6: Update sequence of terminating prog1 and adding prog2.

and allocating it to the data plane at runtime. When a new program arrives, the P4runpro compiler first checks its syntax and semantics, generating the abstract syntax tree (AST). Then, it traverses the AST, performs the address translation steps, and converts pseudo primitives to primitives. Following that, it performs a resource allocation scheme to allocate translated AST nodes to RPBs. Finally, it generates table entries and consistently updates the data plane. Figure 5 illustrates the compilation process of the program cache.

Syntax and Semantics Check. We provide the syntax of P4runpro in Figure 15 in Appendix §B.1 for brevity. The semantics of P4runpro is straightforward, so the compiler only performs a type check on primitive arguments when generating the AST. As shown in Figure 5(a), each primitive in the program cache is translated into an AST node. We omit the first 3 nodes and arguments. The depth of the AST node refers to the primitive execution dependency. Each branch of the AST represents a conditional branch in the P4runpro program.

Primitive Translation. After generating the AST, the compiler performs the address translation steps and pseudo primitive translation. During this process, the compiler also aligns memory primitives that operate on the same virtual memory but in different branches to the same depth, since the hardware does not support cross-stage memory access. As illustrated in Figure 5(b), the compiler introduces the offset step before MEMREAD and MEMWRITE and inserts a "nop" after LOADI in the middle branch to align the memory operations.

Program Allocation. Using a constraint-solving model to find a suitable allocation on the programmable data plane is nontrivial [22, 27, 31, 58], especially at runtime, as there is no mature toolchain, such as P4C, to assist with resource allocation. ActiveRMT [22] focuses solely on memory allocation since it simulates table-matching logic into memory load and comparison, and this results in low generality and high resource usage. ActiveRMT uses a fair worst-fit allocation scheme that remaps memory for elastic programs. In contrast, we enable dynamic table matching logic through BRANCH primitive. Thus, we also need to consider table entry allocation. In addition, we use a first-come-first-serve and best-effort allocation scheme since the switch programs are configured by the operator, instead of a client in ActiveRMT. We leave the trade-off of resource usage between the programs to the operator. In the resource manager, we use bidirectional linked lists to maintain free memory partitions, supporting only continuous memory allocation.

Problem Formulation. We use an SMT model to find an allocation vector $\mathbf{x} \in \{x_i | x_i = 1, ..., M * (R+1)\}^L$, where x_i is the allocated logic RPB number, L is the depth of translated program AST (*e.g.*, $L = 10$ in Figure 5(b)), M is the maximum physical RPB number, and R is the maximum recirculation iteration number. We force the same primitives at the same AST depth executed in the same RPB to reduce complexity. The optimization goal is to minimize the objective function $f(\mathbf{x}) = \alpha x_L - \beta x_1$ where α and β are weight parameters. According to our evaluation, the objective function greatly influences the performance of the P4runpro compiler. We choose the objective function above since it balances the complexity, resource utilization, and program capacity of the allocation scheme (§6.2.4). The formulation is as follows:

$$minimize \;\; \alpha x_L - \beta x_1,$$

$$subject\ to \;\; \bigwedge_{1<=i<=M-1} (x_i + 1 <= x_{i+1}) \quad (1)$$

$$\bigwedge_{1<=i<=M} (te\_req(x_i) <= te\_free(x_i)) \quad (2)$$

$$\bigwedge_{1<=i<=M} (mem\_req(x_i) <= mem\_free(x_i)) \quad (3)$$

$$\bigwedge_{i \in \mathcal{F}} \bigvee_{0<=j<=R} (1 + M * j <= x_i <= N + M * j) \quad (4)$$

$$\bigwedge_{(i,j) \in \mathcal{B}} \bigvee_{1<=k<=R} (x_j = x_i + M * k) \quad (5)$$

where $i \in \mathcal{F}$ if one or more primitives with depth x_i is forwarding primitive and $(i, j) \in \mathcal{B}$ if one primitive with depth x_i and one with depth x_j $(x_i < x_j)$ operate the same virtual memory sequentially. N is the maximum ingress RPB number. $te\_req(x_i)$ maps from x_i to the table entry resource requirement for implementing the primitives of depth x_i, and $te\_free(x_i)$ maps from x_i to the free table entries that the logic RPB x_i has. The mappings of $mem\_req(x_i)$ and $mem\_free(x_i)$ are similar but for memory. These mappings are not linear but can be translated to inequality constraints. Constraint (1) presents the primitive dependency. Constraints (2) and (3) are the table entry and memory constraints. Constraint (4) presents the constraint of forwarding primitives only being executed in ingress RPBs. Constraint (5) presents the constraint of hardware not supporting cross-stage memory access. Figure 5(c) shows the allocation results of the program cache in the situation that all the memory of RPB9 is occupied by other running programs. In this case, the compiler moves the executions of the memory primitives to the next RPB.

Consistent Update. Consistent update is a non-trivial concept when updating the network [48–50, 55]. However, some recent efforts [22, 63] do not concern about it. In this paper, the consistent update refers to ensuring that no incorrectly processed packets are exposed to the traffic. Specifically, when updating the data plane by configuring table entries sequentially, there are intermediate states where some entries are already configured, but others are not. For example, if we first enable `FORWARD` in the program `cache` but do not enable the former `BRANCH`, all cache hit packets will be forwarded to the server. Thanks to the RMT architecture's atomicity guarantee for each single-entry update, we achieve consistency by utilizing the unique program ID per program. We batch the table entries per program in the compiler and update the data plane in batches. Figure 6 illustrates the update process of first terminating `P4runpro` program `prog1` and then adding `prog2`. When deleting a program from the data plane, the filtering rules of the initialization block will be deleted first to disable the program ID setting (②). Without the correct program ID, all the following program components stop working at the same time. Then we gradually delete other entries (③). To prevent memory leaks, during the program termination, we lock and reset the memory used by the program (④). The locked memory remains unavailable for reallocation until the reset is complete. Similarly, when adding the entry batch of a program, we first update other components of the program (⑥, ⑦). At last, we update the initialization block when all the program components are ready (⑧).

5 IMPLEMENTATION

We implement a `P4runpro` prototype on a single pipeline of Intel Tofino [32] programmable switch. Utilizing other ASICs with more pipeline stages (e.g., Tofino2 [33]) or folding multiple pipelines [26, 46] can achieve higher performance. Our control plane runs on the Open Network Linux (ONL) 3.16 with a 4-core CPU and 16GB RAM. We use a 36-core, 376GB RAM Ubuntu 20.04 server equipped with a 100G Intel E810-CQDA2 NIC for traffic generation and analysis.

Data Plane. We implement the `P4runpro` data plane with ≈10.5K line of $P4_{16}$ code. We set the PHV register and memory bucket width to 32, the maximum operable width of hardware ALUs, which can also be set to less than 32. All the data plane blocks are implemented with match-action tables. We implement the initialization block using K tables, where K is the number of possible parsing paths. The parser and the initialization block can be customized according to the needs of the operator. We implement the recirculation block with a specific table rewriting the `P4runpro` headers according to the control flags of program execution. The recirculation block is not indispensable, as it can be replaced by multiple switches processing sequentially. In that case, 1 more RPB is supported and the constraints (4) and (5) in §4.3 need to be adjusted. Each RPB is implemented by a large table with the keys of control flags and registers and the actions implementing the atomic operations. We set 10 ingress and 12 egress RPBs, each with a physical memory size of 65,536 and a table size of 2,048.

Control Plane. We implement the `P4runpro` control plane with ≈2.1K lines of Python code. We implement a runtime CLI to interact with the `P4runpro` data plane. For the `P4runpro` compiler, we use Python Lex-Yacc [7] for scanning and parsing `P4runpro` programs. We employ the Z3 Prover [10] to solve our SMT model of resource allocation. We use the `bfrt_grpc` APIs to update and monitor the data plane.

Program Name	LoC		Update Delay (ms)	
	Ours	P4	Ours	Others
In-network Cache	26	77	11.47	194.30*
Stateless Load Balancer [12]	15	63	10.63	225.46*
Heavy Hitter Detector [52]	36	109	30.64	228.70*
NetCache [35]	60	152	40.06	-
DQAcc [58]	16	137	15.45	-
Stateful Firewall	22	88	19.70	-
L2 Forwarding	10	33	2.98	-
L3 Routing	6	34	1.88	-
Tunnel	6	51	2.38	-
Calculator	26	53	26.74	-
ECN	9	18	4.84	-
Count-Min Sketch (CMS) [21]	14	78	14.21	27.46**
Bloom Filter (BF) [44]	14	78	12.51	32.09**
SuMax [62]	14	80	19.94	22.88**
HyperLogLog (HLL) [24]	167	180	166.90	17.37**

* is ActiveRMT's update delay, and ** is FlyMon's.

Table 1: P4 Programs Implemented by `P4runpro` and the delay for updating them.

Traffic Generation. We use Cisco TRex [20] to generate a large volume of stateless traffic for our evaluation. We tcpreplay [9] and libpcap [1] to replay the real-world traffic and analyze the received packets.

6 EVALUATION

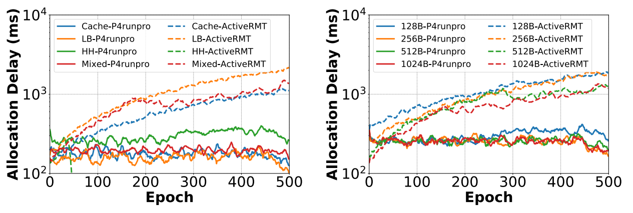
We evaluate the generality and expressiveness, performance, and overhead of our prototype and conduct four case studies compared to conventional P4 programs. We intend to answer the following questions:

- Are `P4runpro` primitives and pseudo primitives general enough to express various P4 programs (§6.1)?
- Does our design have good performance in program deployment delay (§6.2.1), resource utilization (§6.2.2), and program capacity (§6.2.3)?
- Does our design introduce excessive overhead (§6.3)?
- Does the runtime programming impact running traffic and program functions (§6.4)?

6.1 Generality and Expressiveness

To demonstrate `P4runpro`'s generality and expressiveness, we use it to implement 15 conventional P4 programs listed in Table 1. These programs are implemented with reference to the corresponding literature [12, 21, 24, 35, 44, 53, 58, 62], existing P4 implementations [17, 34], and P4 tutorials [5]. They cover a diverse range of applications, including in-network computation, network measurement, and traffic forwarding. The heterogeneity of the programs we implemented goes beyond what ActiveRMT and FlyMon currently support, thereby demonstrating `P4runpro`'s generality[1]. As for expressiveness, we compare the LoC between our `P4runpro` program's complete processing logic and the P4 programs' control

[1]It is difficult for us to prove how many other programs that prior work can support. We only compare the programs they achieve in the artifacts we access in this paper.



(a) Allocation delay during continuous program deployment.

(b) Allocation delay of different memory granularity.

Figure 7: Allocation delay.

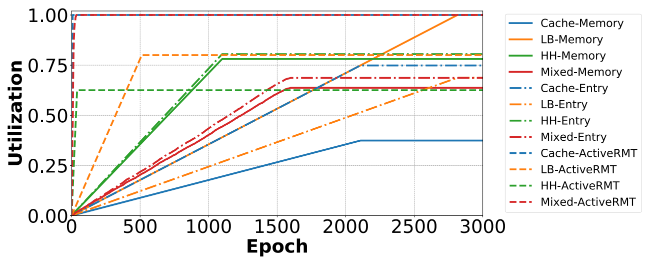


Figure 8: Memory and table entry utilization.

block with equivalent functionality, as shown in Table 1. Note that we only compare the LoC that comprises the packet processing logic here. Elastic case blocks, which do not embody program logic, are excluded from the count. The elastic case blocks of P4runpro program denote the case blocks in conditional branches whose number is variable (*e.g.*, line 10 to line 28 in Figure 2). They correspond to the non-constant table entries in the P4 context and are not expressed in the P4 code. Similarly, inelastic case blocks are necessary for data plane packet processing logic, corresponding to the constant table entries or if-else statements in P4. Also, the fixed header definition and (de)parsing logic of P4 programs are not included. The comparison results highlight P4runpro's ability to express complex P4 logic more simply.

6.2 Performance

We evaluate our prototype's performance and compare it to ActiveRMT. To ensure fairness, we enable ActiveRMT's least constraint allocation model with a memory size of 65,536, and we set our prototype's maximum recirculation iteration number R to 1. The weights α and β are set to 0.7 and 0.3, respectively. We will explain how we decide the recirculation iteration number R and the objective function parameters later. Unless otherwise specified, the memory size of programs is 1,024B, *i.e.*, 256 32-bit buckets.

6.2.1 Program Deployment Delay. We evaluate the program deployment delay of P4runpro to illustrate the benefit of runtime programmability. In the conventional P4 workflow, compiling a P4 program can take a few or even a dozen minutes, and switch reprovisioning takes a few seconds. Using P4runpro, the program deployment delay can be reduced by at least one order of magnitude, all without disturbing traffic and other programs. The deployment delay primarily includes two components: update delay and allocation delay, as the program parsing delay is negligible according to our tests (average ≈2 ms).

Update Delay. The update delay presented in Table 1 represents the time required for data plane updating. We conduct tests for the average update delay over 50 repeated updates for each program. The results indicate a positive correlation between update delay and program complexity. Due to the large number of inelastic case blocks in HLL, it experiences a worse update delay. Compared to FlyMon and ActiveRMT, P4runpro exhibits faster update times for most programs.

Allocation Delay. The allocation delay represents the time required for the allocation scheme computation. The comparison of our allocation delay with ActiveRMT reveals that P4runpro performs better in most cases. We choose workload programs for in-network cache (cache), stateless load balancer (lb), heavy hitter detector (hh), and a mix of them. The mixed workload means randomly choosing one of the three programs to deploy in each epoch. We arrange 500 arrivals of these workloads sequentially and record the allocation delay. When allocation fails, the allocation time is set to 0. We repeat this 10 times, and the moving average results with the window size of 31 are shown in Figure 7(a). We observe that the allocation delay of P4runpro is relatively stable per epoch but varies among different programs. This variation is because the complexity of our allocation scheme is not sensitive to the number of allocated resources but increases with the depth of the input AST. The allocation delay of P4runpro also fluctuates in the short term since the constraints for each deployment are not the same, given changes in resource usage. In contrast, with an increasing number of allocated programs, ActiveRMT experiences a rapid increase in allocation time, exceeding 1 second. In addition, ActiveRMT only supports fixed memory allocation granularity, and finer granularity leads to an increase in allocation delay. P4runpro supports more flexible memory sizes in powers of 2. We test the allocation delay with different memory granularities ranging from 128B to 1,024B under the mixed workload. As shown in Figure 7(b), the requested memory size does not affect the allocation time of P4runpro.

6.2.2 Resource Utilization. In our design, we simultaneously utilize the memory and table entries. To evaluate resource utilization and compare to ActiveRMT, we continuously allocate memory and table entries for programs until failure. Figure 8 depicts the results under the same workloads mentioned above in §6.2.1. ActiveRMT treats the program cache as an elastic program, allowing its memory to be subtracted for new programs. As a result, the cache and mixed workload can reach maximum utilization. For the lb and hh workloads, P4runpro eventually has higher resource utilization and program capacity. Overall, the resource utilization under these workloads ranges from 60% and 80%. We further analyzed the reasons why the program could not continue to be allocated. We find that P4runpro's memory utilization is limited for the cache and hh workloads due to primitive dependencies, preventing them from achieving full utilization. Specifically, since the forwarding primitives in the program cache and hh must be executed after several other primitives, all the free entries of ingress RPBs are occupied after multiple allocations. However, forwarding primitives can only be executed in the ingress RPB, causing allocation failure. In contrast, for the simpler program lb, the allocation scheme reaches 100% of memory allocation.

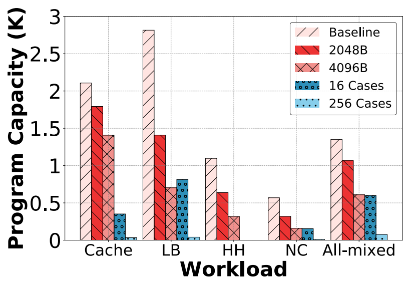


Figure 9: Program capacity.

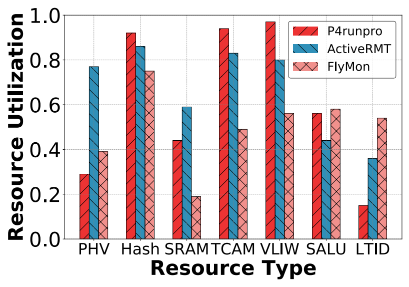


Figure 10: Resource overhead.

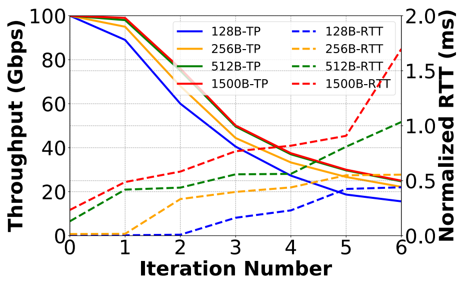
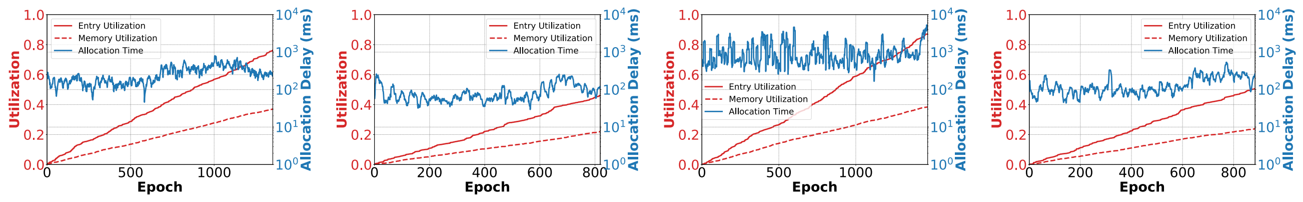


Figure 11: Recirculation overhead.



(a) $\alpha x_L - \beta x_1$. (b) x_L. (c) $\frac{x_L}{x_1}$. (d) Hierarchical optimization.

Figure 12: Performance of different objective functions under the all-mixed workload.

6.2.3 Program Capacity. As shown in Figure 9, we use program capacity to measure the concurrency of how many programs can run simultaneously. We evaluate the previous three workloads of `cache`, `lb`, and `hh`, the most complex program NetCache (`nc`) of the 15 programs, and the all-mixed workload that includes all the 15 programs in Table 1. The baseline is derived from the same setup above in which each program has 1,024B memory and 2 elastic case blocks if applicable. Subsequently, based on this setup, we enhance the requested memory size to 2,048B and 4,096B, or the elastic case block number to 16 and 256. Note that some programs, such as `hh`, do not have elastic case blocks. The results reveal that, for the simpler program `lb`, `P4runpro` can achieve a program capacity of ≈2.8K. Similarly, for the most complex program `nc`, `P4runpro` attains a program capacity of ≈0.6K. Under the all-mixed workload, `P4runpro` achieves a program capacity ranging from 77 to 1351 based on the resource requests. In addition, we find that doubling the memory size does not halve the program capacity; instead, the increase in elastic block number has a more significant impact on program capacity. The table entry resource in `P4runpro` is more scarce than the memory resource due to much less in-switch TCAM than SRAM.

6.2.4 Alternative Optimization Schemes. To keep generality in allocation for customized programs, we do not use a program-specific allocation scheme like ActiveRMT. The objective function is the main reason that affects allocation performance. Throughout the above experiment, we find that the primary cause for allocation failure is the non-uniform entry allocation. We optimize this by allocating primitives on the egress RPBs as much as possible (*i.e.*, maximizing x_1) while striving to avoid recirculation (*i.e.*, minimizing x_L). We evaluate the performance of three objective functions $f_1(\mathbf{x}) = \alpha x_L - \beta x_1$ with $\alpha = 0.7$ and $\beta = 0.3$, $f_2(\mathbf{x}) = x_L$, and $f_3(\mathbf{x}) = \frac{x_L}{x_1}$ and the hierarchical optimization which first minimizes x_L and then maximizes x_1 using the value of x_L obtained in the first step. The values of α and β are obtained by our pre-experiment with the parameter sweep analysis of step size 0.1. We continuously deploy the all-mixed workload using the four schemes until failure. The results are shown in Figure 12. The function $f_2(\mathbf{x}) = x_L$ and the hierarchical optimization scheme have the lowest program capacity (the length of the X-axis) and resource utilization, and the hierarchical optimization scheme has a higher allocation delay than $f_2(\mathbf{x}) = x_L$ due to the two-step computing. The function $f_1(\mathbf{x}) = \alpha x_L - \beta x_1$ has the second highest resource utilization, and the allocation delay averages a few hundred milliseconds. The function $f_3(\mathbf{x}) = \frac{x_L}{x_1}$ achieves the largest program capacity and resource utilization but the allocation delay is up to 1 or even 10 seconds due to the nonlinear objective function. The memory utilization is relatively low in all the schemes since 5 of the 15 programs do not need to request memory. To balance the allocation delay, resource utilization, and program capacity, we finally choose the function $f_1(\mathbf{x}) = \alpha x_L - \beta x_1$ as the objective function of our prototype. The detailed comparison and analysis are shown in Appendix §C. It is challenging to prove mathematically what objective function is optimal, and the distribution of the programs we input does not represent the real-world distribution. Thus during the operation of `P4runpro`, the objective function should be empirically adjusted according to the distribution of input programs.

6.3 Overhead

Resource, Latency, and Power. We use P4C [2] and P4 Insight [3] to calculate the resource, latency, and power usage of our prototype and compare the results to those of ActiveRMT and FlyMon. We show the usage of seven main resources, *i.e.*, PHV, hash unit, SRAM, TCAM, VLIW, SALU, and logical table ID (LTID), in Figure 10. We find that `P4runpro` efficiently utilizes the PHV and LTID and `P4runpro` uses almost all the VLIW to implement atomic operations. The usage of the hash unit and SALU exceeds that of ActiveRMT due to the inclusion of two extra RPB stages in `P4runpro`. `P4runpro` does not heavily rely on SRAM, and the unused SRAM can be

	Latency (cycles)	Power (W)	Load
P4runpro	306/316/622	19.32/21.42/40.74	98%
ActiveRMT	312/308/620	23.36/20.34/43.7	91%
FlyMon	54/282/336	0/34.05/34.05	100%

Table 2: Latency added by clock cycle, worst-case power, and traffic limit load simulated by P4C. The data separated by the "/" sign represents ingress/egress/total respectively.

leveraged to scale the memory size. However, TCAM usage limits the scalability of the table size per RPB. The results of latency and power usage are shown in Table 2. Compared to ActiveRMT, P4runpro has almost the same latency but lower worst-case power consumption. Due to the exceeding of the hardware's power budget (40.00W), ActiveRMT introduces a traffic limit load of 91% which means that the hardware will limit the forwarding rate to 91% of the maximum, introducing throughput loss. For P4runpro, the traffic limit load is 98%. FlyMon is tied to the scope of the network measurement tasks. Therefore it does not need to introduce extra resources, latency, and power overhead to ensure generality.

Impact of Recirculation. Enabling recirculation leads to both throughput loss and increased latency. In Figure 11, we evaluate the impact of recirculation on throughput loss and latency increase across different packet sizes, ranging from 128B to 1500B. For throughput evaluation, we generate traffic up to 100Gbps from one switch port to another. We record the maximum throughput without packet loss for different recirculation iteration numbers. Regarding latency, we send packets through the switch and record the average zero-queue RTT. The RTT is normalized by the minimum RTT to better display the RTT increment. The results show that with only one recirculation iteration, the throughput loss is acceptable (ranging from 1% to 10%, depending on the packet size). Thanks to the high-rate processing pipeline of the switch, the additional latency brought by recirculation is minimal—only 0.5 to 1.5ms (2.2% to 7.2% of growth rate) even when the recirculation iteration number is 6. In our prototype, we set the maximum recirculation iteration number $R = 1$, as it incurs manageable overhead while providing sufficient flexibility. All the programs we evaluate can be processed within 1 recirculation iteration. Most of them (13 of 15) can be processed without recirculation. In scenarios where there is no heavy throughput, the number of recirculations can be relaxed to a larger number to support longer programs and loosen the compiler's allocation constraints.

6.4 Case Study

To illustrate the process of runtime programming and validate that P4runpro does not impact running traffic and program functions, we conducted case studies using the P4runpro programs, comparing them to conventional P4 programs. The P4runpro programs of lb and hh are listed in Appendix §B.2. We use a dataset comprising ≈1.3GB of TCP and UDP traffic mirrored from the Tsinghua University campus network on a weekend afternoon.

Ethical Consideration. We processed the traffic with the consent and supervision of the campus network operator. The traffic was anonymized by the campus network operator before being accessed by researchers, ensuring the confidentiality of personal data. The MAC and IP addresses underwent a one-way hash mapping to fake addresses. All packet payloads were replaced with duplicated identical bytes.

Setup. We further processed the traffic, setting the 5-tuple to 4,096 distinct combinations. In terms of the dataset of the in-network cache workload, we extracted the UDP packets from the raw dataset, edited the destination port to the same, discarded the payload, and attached a cache header to it. We replayed the traffic at 100Mbps and collected data per 50ms. For all case studies, we started to deploy the P4runpro programs and conventional P4 programs at 5s.

Impacts on Traffic. We run a conventional P4 program with only a forwarding table and our prototype with a basic forwarding program at the beginning. At time 5s we start deploying and deleting P4runpro programs in Table 1 at the frequency of random one per 0.5s. These program's filtering rules are set independently of the traffic to avoid impacts from the program logic itself. We record the RX rate of P4runpro and the contrast traffic in Figure 13(a). The results show that the runtime deployment of switch programs does not impact the running traffic. The spikes in the curve are caused by large TCP packet transfers as the results in the case study of in-network cache using UDP packets without payload do not have spikes.

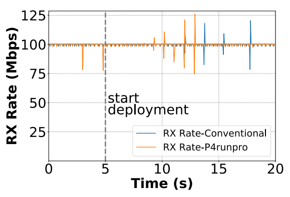
In-network Cache. We start deploying the program cache using P4 program and on P4runpro at the same time in Figure 13(b). We set the cache keys according to the ground truth to keep the hit rate to 0.6. We start deploying programs at the same time. Compared to the conventional P4 program, P4runpro has almost no deployment delay and runs the program's function faster than the conventional program since there is no need to reprovision the switch and enable ports. In addition, the two programs have no difference in program function. The hit rate is 0.6, therefore 60Mbps of cache read traffic is reflected and the RX rate is 40Mbps.

Stateless Load Balancer. We start deploying the program lb using P4 program and on P4runpro at the same time in Figure 13(c). The DIPs are distributed to two ports. We record the load imbalance rate through $\frac{|rx\_rate\_port\_1 - rx\_rate\_port\_2|}{total\_rx\_rate}$. The result shows that there is also no difference between the two programs' functions except for deployment delay.

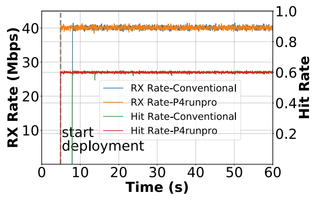
Heavy Hitter Detector. In the program hh, P4runpro uses the mask step of address translation. Prior work [63] proves that if the hash algorithms are perfectly uniform, truncating the hash algorithm with a high output width has the same collision probability as what with the same lower output width. The standard hash algorithms fit this condition very well, and our results in Figure 13(d) illustrate that. We use the standard hash algorithms crc_16_buypass, crc_16_mcrf4xx, crc_aug_ccitt, and crc_16_dds_110 to calculate memory addresses of 2 rows of CMS and BF. We set both the memory size and threshold as 1024. We set 100 flows that appear more than 1024 times as the ground truth. We use the F1 score to evaluate the accuracy of the two programs. The result shows that P4runpro has the same function and performance as the conventional program. After a while, the F1 score rapidly reaches 1, indicating that the programs effectively detect all high-frequency flows without any misreport.

7 DISCUSSION AND LIMITATIONS

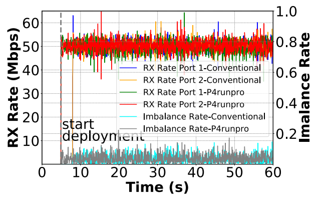
Header Parsing. The RMT ASICs do not support runtime reconfiguration of the parsing state machine. Hence, we cannot enable



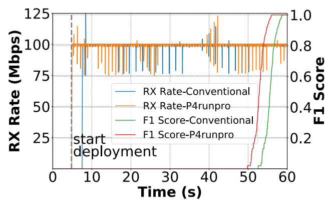
(a) Impacts on traffic.



(b) In-network cache.



(c) Stateless load balancer.



(d) Heavy hitter detector.

Figure 13: Case studies.

dynamic header parsing logic. The runtime programming is under the scope of predefined header parsing logic.

Memory Allocation. The mask-based address translation mechanism enables only memory sizes that are powers of two. In addition to that, we only use continuous free memory for allocation. Though the TCAM-based address translation [63] can enable granular memory size and partition memory allocation, it brings huge TCAM and VLIW consumption that is unacceptable in our design. Similar to the memory allocation in OSes, a continuous memory allocation algorithm with powers of two generates both external and internal memory fragmentations, reducing memory utilization. Enable the *direct* mapping mechanism proposed by SwitchVM [37] in P4runpro can help utilize these fragmentations. This optimization is possible, but it requires manually deciding which memory translation mechanism to use for each program since there is no compilation support for it.

Parallel Execution. Our design has not yet incorporated parallel program execution for the same flow. The unrelated switch programs operating the same packets need to be merged in a single program using BRANCH for parallelism or executing sequentially.

Incremental Update. Incremental updates of running programs have not been supported well in the P4runpro control plane so far. For example, when adding a new key-value pair to the program cache, two additional case blocks must be embedded within the program and then updated to the data plane. We implement it by manually updating the corresponding data plane tables or invoking the P4runpro compiler to revoke the old one and allocate the new one. We consider optimizing the incremental update as future work to enrich control plane functionality.

Entry Expansion. To ensure generality, all the tables in P4runpro use ternary match and have redundant keys, reducing the table size. This is not suitable for lookup-intensive programs that require mainly table lookups instead of computation. However, these programs are less likely to be deployed on programmable switches since they are well-supported by traditional fixed-function switches with much larger entry capacities.

Generality and Expressiveness. As discussed in Section §4.2, the capacity of header interaction primitives is the main limitation on the generality of P4runpro. For the hash operation, we can use XOR to generate the hash output of multiple customized fields [63]. The combination of primitives SGT, SLT, EQUAL, and BRANCH can express complex conditions of comparison and equality. Due to the range match supporting up to 20-bit key, we do not use it when implementing primitive BRANCH in our prototype that directly simulates the condition branch in P4. Unfortunately, we cannot support shift operations due to the VLIW constraint. In addition to the programs we evaluated, we try to consider the in-network aggregation programs such as SwitchML [51] and ATP [40]. Implementing the simple aggregation logic in SwitchML requires only modifying P4runpro to support multicast. We failed to implement ATP using P4runpro primitives due to its complicated logic.

Flexibility v.s. Performance. A trade-off is between flexibility and performance in P4runpro. To enable dynamic memory and support more operations, we use multiple stages to implement certain single-stage operations in the P4 context, which increases the flexibility of runtime programming but results in more stage usage and possible recirculation. Due to the ~Tbps high bandwidth of programmable switches, their throughput is usually wasted. We believe that switches at different positions of the network have different requirements. For the switches deployed in a high-throughput position such as cloud gateway [46], the operator needs to deploy huge forwarding tables and disable recirculation to achieve full throughput. In the scenario with relatively low throughput (*e.g.*, the leaf switch connected with servers), however, the operator can deploy P4runpro to gain the benefit of flexible isolated programming and low response time. Additionally, compared to prior work, P4runpro introduces less overhead and achieves higher generality and performance.

8 CONCLUSION

In this paper, we propose P4runpro. P4runpro enables runtime programmability for RMT switches with enough generality, good performance, and acceptable overhead. We believe that P4runpro is helpful for not only flexible programming on existing RMT ASICs but also the design and development of new programmable network devices. We hope that one day programmable networks will have a real operating layer or hypervisor to abstract programming from the hardware. Our future work includes the following two parts. First, we intend to make the P4runpro compiler a back end of P4C, directly updating P4 programs to the data plane at runtime. Second, we want to enrich the control plane functionality.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Marios Kogias and the anonymous reviewers for their insightful comments and helpful feedback. We also appreciate the support from the Internet Measurement and Analysis Data Sharing Platform for providing campus network traffic that enabled us to conduct case studies. This work is supported by the National Natural Science Foundation of China under Grant No. 62302253 and 61772307, the Zhongguancun Laboratory Project, and the Beijing Natural Science Foundation under Grant No. 4222026. Lin He is the corresponding author of this paper.

REFERENCES

[1] 2024. libpcap. https://github.com/the-tcpdump-group/libpcap.

[2] 2024. P4 Compiler. https://github.com/p4lang/p4c.

[3] 2024. P4 Insight. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.

[4] 2024. P4 language. https://p4.org.

[5] 2024. P4 tutorials. https://github.com/p4lang/tutorials.

[6] 2024. P4runpro prototype implementation. https://github.com/P4runpro/P4runpro.

[7] 2024. Python Lex-Yacc. https://github.com/dabeaz/ply.

[8] 2024. RISC-V Instruction Set Architecture. https://riscv.org.

[9] 2024. tcpreplay. https://github.com/appneta/tcpreplay.

[10] 2024. Z3 Prover. https://github.com/Z3Prover/z3.

[11] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 51–70. https://www.usenix.org/conference/nsdi22/presentation/addanki

[12] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[15] Jiamin Cao, Ying Liu, Yu Zhou, Lin He, Chen Sun, Yangyang Wang, and Mingwei Xu. 2022. CoFilter: High-Performance Switch-Accelerated Stateful Packet Filter for Bare-Metal Servers. *IEEE Transactions on Parallel and Distributed Systems* 33, 9 (2022), 2249–2262. https://doi.org/10.1109/TPDS.2021.3136575

[16] Jiamin Cao, Ying Liu, Yu Zhou, Lin He, and Mingwei Xu. 2022. TurboNet: Faithfully Emulating Networks With Programmable Switches. *IEEE/ACM Transactions on Networking* 30, 3 (2022), 1395–1409. https://doi.org/10.1109/TNET.2022.3142126

[17] Xiaoqi Chen. 2024. HyperLogLog P4 implemetation. https://github.com/Princeton-Cabernet/p4-projects/tree/master/HyperLogLog-tofino.

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

[19] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. DRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823

[20] Cisco. 2024. TRex traffic generator. https://trex-tgn.cisco.com.

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

[22] Rajdeep Das and Alex C Snoeren. 2023. Memory Management in ActiveRMT: Towards Runtime-Programmable Switches. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1043–1059. https://doi.org/10.1145/3603269.3604864

[23] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. 2022. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 635–649. https://www.usenix.org/conference/nsdi22/presentation/feng

[24] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) (2007).

[25] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An Open-Source 100-Gbps Nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 38–46. https://doi.org/10.1109/FCCM48280.2020.00015

[26] Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai. 2024. Sirius: Composing Network Function Chains into P4-Capable Edge Gateways. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 477–490. https://www.usenix.org/conference/nsdi24/presentation/gao-jiaqi

[27] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 44–61. https://doi.org/10.1145/3387514.3405852

[28] David Hancock and Jacobus van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/2999572.2999607

[29] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2023. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications* 212 (2023), 103561. https://doi.org/10.1016/j.jnca.2022.103561

[30] Lin He, Peng Kuang, Ying Liu, Gang Ren, and Jiahai Yang. 2021. Towards Securing Duplicate Address Detection using P4. *Computer Networks* 198 (2021), 108323. https://doi.org/10.1016/j.comnet.2021.108323

[31] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[32] Intel. 2024. Intel tofino series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[33] Intel. 2024. Intel tofino2 series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html.

[34] Xin Jin. 2024. NetCache P4 implemetation. https://github.com/netx-repo/netcache-p4.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136.

[36] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 595–612.

[37] Sajy Khashab, Alon Rashelbach, and Mark Silberstein. 2024. Multitenant In-Network Acceleration with SwitchVM. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 691–708. https://www.usenix.org/conference/nsdi24/presentation/khashab

[38] Hyojoon Kim, Xiaoqi Chen, Jack Brassil, and Jennifer Rexford. 2021. Experience-Driven Research on Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 51, 1 (mar 2021), 10–17. https://doi.org/10.1145/3457175.3457178

[39] Fadi J Kurdahi and Alice C Parker. 1987. REAL: A program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*. 210–215.

[40] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao

[41] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. 2022. IMap: Fast and Scalable In-Network Scanning with Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 667–681.

[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 44–58. https://doi.org/10.1145/3341302.3342085

[43] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow

Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[44] Yuxin Meng and Lam-For Kwok. 2014. Adaptive Blacklist-Based Packet Filter with a Statistic-Based Approach in Network Intrusion Detection. *J. Netw. Comput. Appl.* 39, C (mar 2014), 83–92.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[46] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3452296.3472889

[47] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 5 (1999), 895–913.

[48] Yiming Qiu, Ryan Beckett, and Ang Chen. 2023. Synthesizing Runtime Programmable Switch Updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 613–628. https://www.usenix.org/conference/nsdi23/presentation/qiu

[49] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Helsinki, Finland) *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 323–334. https://doi.org/10.1145/2342356.2342427

[50] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent Updates for Software-Defined Networks: Change You Can Believe In!. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (Cambridge, Massachusetts) *(HotNets-X)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2070562.2070569

[51] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[52] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3050220.3063772

[53] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2407–2422. https://doi.org/10.1145/3318464.3389698

[54] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. 2022. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1289–1305. https://www.usenix.org/conference/nsdi22/presentation/wang-tao

[55] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 651–665.

[56] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. 2022. Bedrock: Programmable Network Support for Secure RDMA Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2585–2600.

[57] Jiarong Xing, Qiao Kang, and Ang Chen. 2020. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2039–2056.

[58] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, and Bin Liu. 2023. ClickINC: In-Network Computing as a Service in Heterogeneous Programmable Data-Center Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 798–815. https://doi.org/10.1145/3603269.3604835

[59] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized Network Snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 402–416. https://doi.org/10.1145/3230543.3230552

[60] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1345–1358. https://www.usenix.org/conference/nsdi22/presentation/zeng

[61] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. *2017 26th International Conference on Computer Communication and Networks (ICCCN)* (2017), 1–9.

[62] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 991–1010. https://www.usenix.org/conference/nsdi21/presentation/zhao

[63] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: Enabling on-the-Fly Task Reconfiguration for Network Measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 486–502.

[64] Peng Zheng, Theophilus A. Benson, and Chengchen Hu. 2018. P4Visor: lightweight virtualization and composition primitives for building and testing modular programs. *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies* (2018).

[65] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2020. Newton: Intent-Driven Network Traffic Monitoring. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3386367.3431298

[66] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, Anirudh Sivaraman, and Xin Jin. 2022. NetVRM: Virtual Register Memory for Programmable Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 155–170. https://www.usenix.org/conference/nsdi22/presentation/zhu

[67] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (2014), 32–41. https://doi.org/10.1109/MM.2014.61

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A P4RUNPRO PRIMITIVE

A.1 `P4runpro` Primitive and Pseudo Primitive Set

Table 3 enumerates all the primitives and pseudo primitives along with the expressions of corresponding data plane operations supported by `P4runpro`. Table 4 enumerates all the argument types and takes the program `cache` as an example.

A.2 Pseudo Primitive Translation

Figure 14 shows the translation of all pseudo primitives. We use `LOADI` and `ADD` to express `MOVE`. We use `LOADI` and `ADD/AND/XOR` to express `ADDI/ANDI/XORI`. We use `LOADI` to load the `m = 0xffffffff` and `XOR` to perform an XOR operation on the target register and `m` to express `NOT`. We use `XOR`, `MIN`, `MAX` to express the comparison primitives `EQUAL`, `SGT`, and `SLT`. For the primitives `SUB`, we first get the one's complement of subtrahend using the same way implementing `NOT`. Then we first add the one's complement to the minuend and use `XOR` to recover the value of the subtrahend. Then we add 1 to the minuend and get the correct value. As for

Type	Primitive	Operation	Explanation
Header interaction	`EXTRACT(field, reg)`	`reg = field`	`field`: header field
	`MODIFY(field, reg)`	`field = reg`	`reg`: register
hash	`HASH_5_TUPLE`	`har = hash(5_tuple)`	`mid`: memory identifier `width`: width of mid size
	`HASH`	`har = hash(har)`	
	`HASH_5_TUPLE_MEM(mid)`	`mar = (bit<width>)hash(5_tuple)`	
	`HASH_MEM(mid)`	`mar = (bit<width>)hash(har)`	
Conditional branch	`BRANCH`	`branch_id = bid(mar, sar, har)`	`bid`: branch ID

Memory	`MEMADD(mid)`	`mid[mar] = mid[mar] + sar`
		`sar = mid[mar]`	

	`MEMSUB(mid)`	`mid[mar] = mid[mar] - sar`
		`sar = mid[mar]`	

	`MEMAND(mid)`	`mid[mar] = mid[mar] & sar`
		`sar = mid[mar]`	

	`MEMOR(mid)`	`sar = mid[mar]`
		`mid[mar] = mid[mar] | sar`	

	`MEMREAD(mid)`	`sar = mid[mar]`	
	`MEMWRITE(mid)`	`mid[mar] = sar`	
	`MEMMAX(mid)`	`mid[mar] = sar if sar > mid[mar]`	
Arithmetic and logic	`LOADI(reg, i)`	`reg = reg + i`	`i`: immediate operand
	`ADD(reg0, reg1)`	`reg0 = reg0 + reg1`	
	`AND(reg0, reg1)`	`reg0 = reg0 & reg1`	
	`OR(reg0, reg1)`	`reg0 = reg0 | reg1`	
	`MAX(reg0, reg1)`	`reg0 = max(reg0, reg1)`	
	`MIN(reg0, reg1)`	`reg0 = min(reg0, reg1)`	
	`XOR(reg0, reg1)`	`reg0 = reg0 ^ reg1`	
	`MOVE(reg0, reg1)`	`reg0 = reg1`	pseudo primitives
	`NOT(reg)`	`reg = ~reg`	
	`SUB(reg0, reg1)`	`reg0 = reg0 - reg1`	
	`EQUAL(reg0, reg1)`	`reg0 = 0 if reg0 == reg1`	
	`SGT(reg0, reg1)`	`reg0 = 0 if reg0 >= reg1`	
	`SLT(reg0, reg1)`	`reg0 = 0 if reg0 <= reg1`	
	`ADDI(reg, i)`	`reg = reg + i`	
	`ANDI(reg, i)`	`reg = reg & i`	
	`XORI(reg, i)`	`reg = reg ^ i`	
	`SUBI(reg, i)`	`reg = reg - i`	
Forwarding	`FORWARD(port)`	send the packet to `port`	`port`: egress port
	`DROP`	drop the packet	
	`RETURN`	reflect the packet	
	`REPORT`	send the packet to CPU	

Table 3: `P4runpro` primitive set

Type	Example	Explanation
Field	`hdr.nc.key`	header or metadata field
Identifier	`mem1`	memory identifier
Immediate operand	512	32-bit unsigned integer
Register	`har`	hash register
	`mar`	memory address register
	`sar`	stateful ALU register

Table 4: Argument types of `P4runpro` primitives taking the program cache as an example.

`SUBI`, we simply load the immediate operand $m - i + 1$ since it is set by the control plane and add it to the minuend.

B P4RUNPRO PROGRAM

B.1 Syntax

Figure 15 shows the syntax of `P4runpro` language. The symbol '$*$' represents zero, one, or more occurrences, '+' represents one or more occurrences, and '|' represents multiple derivations of a non-terminal. The italicized words represent non-terminals and the other letters, and symbols except for '$*$', '+', and '|' represent terminals. The bold words represent a set of terminals. The terminal set **FIELD** refers to a header or intrinsic metadata field. **VALUE**

```
MOVE(A,B)   = LOADI(A,0)
              ADD(A,B)

ADDI(A,i)   = LOADI(C,i)
              ADD(A,C)

ANDI(A,i)   = LOADI(C,i)
              AND(A,C)

XORI(A,i)   = LOADI(C,i)
              XOR(A,C)

NOT(A)      = LOADI(C,m)
              XOR(A,C)

EQUAL(A,B)  = XOR(A,B)

SGT(A,B)    = MIN(A,B)
              XOR(A,B)

SLT(A,B)    = MAX(A,B)
              XOR(A,B)

SUB(A,B)    = LOADI(C,m) //A-B=A+(m-B+1)
              XOR(B,C)
              ADD(A,B)
              XOR(B,C)
              LOADI(C,1)
              ADD(A,C)

SUBI(A,i)   = LOADI(C,m-i+1)
              ADD(A,C)
```

Figure 14: Translations of pseudo primitives. `A` and `B` is the operation registers, `C` is the supportive register. `m` is the maximum value of a register.

start ::= *annotation∗ program+*
annotation ::= @ **IDENTIFIER INT**
program ::= program **IDENTIFIER** (*filter* ,*filter∗*) { *primitive∗* }
filter ::= < **FIELD** , **VALUE** , **MASK** >
primitive ::= BRANCH : *case+* ;
| **PRIMITIVE_WITH_ARG** (*argument* ,*argument∗*) ;
| **OTHER_PRIMITIVE** ;
case ::= (*condition* ,*condition+*) { *primitive∗* }
condition ::= < **VALUE** , **MASK** >
argument ::= **FIELD** | **IDENTIFIER** | **REGISTER** | **INT**

Figure 15: `P4runpro` syntax.

refers to a binary, decimal, or hexadecimal integer or an IP address. **MASK** refers to a hexadecimal mask. **IDENTIFIER** refers to a program or virtual memory block identifier. **REGISTER** refers to `har`, `mar`, or `sar`. **INT** refers to a binary, decimal, or hexadecimal integer.

B.2 Evaluated Programs

In this section, we list the main evaluated programs, including the stateless load balancer and the heavy hitter detector. The program NetCache is a combination of the in-network cache and heavy hitter detector.

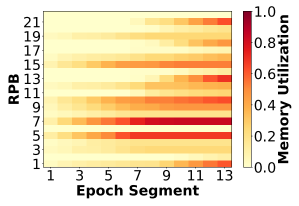
Stateless Load Balancer. Figure 16 illustrates the program stateless load balancer. Instead of using `BRANCH` to check cache hit and packet type in the program cache, We use two memory arrays to store the DIPs and ports to reduce resource usage. These memory buckets can be updated using the raw APIs after the program is allocated. The program first uses primitive `HASH_5_TUPLE_MEM` to locate the buckets of the two memory arrays. Then it reads the bucket storing the port and uses `BRANCH` and `FORWARD` to specify the egress port. It also reads the bucket storing the DIP and then modifies the destination IP address with the DIP.

```
@ dip_pool 1024
@ port_pool  1024
program lb(
    /*filtering traffic*/
    <hdr.ipv4.dst, 10.0.0.0, 0xffff0000>) {
    HASH_5_TUPLE_MEM(port_pool); //locate bucket
    MEMREAD(port_pool); //get egress port
    BRANCH:
    case(<sar, 0, 0xfffffff>) {
        FORWARD(0);
    }
    case(<sar, 1, 0xfffffff>) {
        FORWARD(1);
    };
    MEMREAD(dip_pool); //get DIP
    MODIFY(hdr.ipv4.dst, sar); //write DIP
}
```

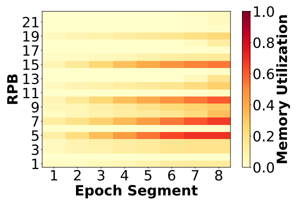
Figure 16: `P4runpro` program for stateless load balancer.

```
@ mem_cms_row1 1024 //CMS with two rows
@ mem_cms_row2 1024
@ mem_bf_row1 1024 //BF with two rows
@ mem_bf_row2 1024
program hh(
    /*filtering traffic*/
    <hdr.ipv4.src, 10.0.0.0, 0xffff0000>) {
    LOADI(sar, 1);
    HASH_5_TUPLE_MEM(mem_cms_row1);
    MEMADD(mem_cms_row1); //count packet
    LOADI(har, 1024); //set threshold
    MIN(har, sar); //compared with threshold
    LOADI(sar, 1);
    HASH_5_TUPLE_MEM(mem_cms_row2);
    MEMADD(mem_cms_row2);
    MIN(har, sar);
    BRANCH:
    /*same flow # exceeds the threshold*/
    case(<har, 1024, 0xffffffff>) {
        LOADI(sar, 1);
        HASH_5_TUPLE_MEM(mem_bf_row1);
        MEMOR(mem_bf_row1); //check existence
        BRANCH:
        /*exist*/
        case(<sar, 1, 0xffffffff>) {
            HASH_5_TUPLE_MEM(mem_bf_row2);
            MEMOR(mem_bf_row2); //check another
            BRANCH:
            case(<sar, 0, 0xffffffff>) {
                REPORT; //report this packet
            };
        }
        /*not exist*/
        case(<sar, 0, 0xffffffff>) {
            LOADI(sar, 1);
            HASH_5_TUPLE_MEM(mem_bf_row2);
            MEMOR(mem_bf_row2); //update another
            REPORT; //report this packet
        };
    };
}
```

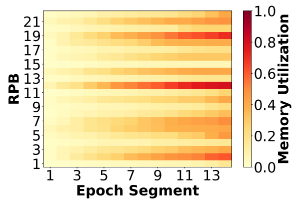
Figure 17: `P4runpro` program for heavy hitter detector.



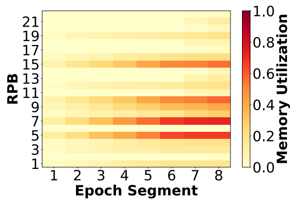
(a) $\alpha x_L - \beta x_1$.



(b) x_L.

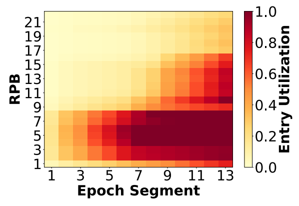


(c) $\frac{x_L}{x_1}$.

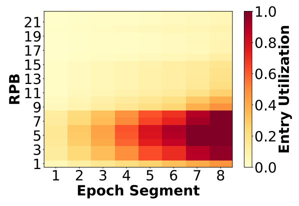


(d) Hierarchical optimization.

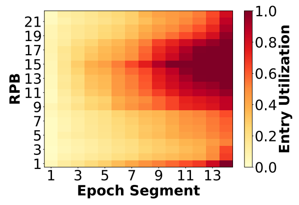
Figure 18: Memory utilization per stage.



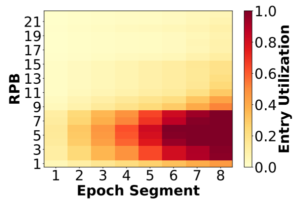
(a) $\alpha x_L - \beta x_1$.



(b) x_L.



(c) $\frac{x_L}{x_1}$.



(d) Hierarchical optimization.

Figure 19: Table entry utilization per stage.

Heavy Hitter Detector. Figure 17 illustrates the program heavy hitter detector. We use a 2-row CMS and a 2-row BF to measure flow frequency and report the high-frequency packet to the control plane. It first uses primitives `LOADI`, `HASH_5_TUPLE_MEM`, and `MEMADD` (line 8 to line 10) to locate the memory bucket of the first row of CMS and adds 1 to it. The result of `MEMADD` is stored in `sar`. Then the program load threshold `1024` to `har` and uses `MIN` to get the minimum value of `har` and `sar` and store it in `har` (line 11 to line 12). The same primitives are executed for the second row of CMS (line 13 to line 16). Then `BRANCN` is used to check the value of `har`. If it equals the threshold, the two memory buckets' values are both greater than the threshold and the flow is considered as a high-frequency flow. Then the program uses `LOADI`, `HASH_5_TUPLE_MEM`, and `MEMOR` to locate the memory bucket of the first row of BF and check the existence (line 20 to line 22). If the memory value before the `MEMOR` is `0`, the flow is first detected and needs to be reported to the control plane (line 34 to line 40). Otherwise, the bucket of the second row of BF is checked to avoid hash collision (line 25 to line 32).

C ALTERNATIVE OPTIMIZATION SCHEMES

During the evaluation of resource utilization and program capacity, we find that the most common reason for allocation failure is primitive dependency. Specifically, the table entries of ingress RPBs run out, leading to that the forwarding primitives cannot be allocated. The egress RPBs' utilization is very low in this situation. To illustrate the processing of memory and table entry allocation per RPB, we draw the heatmaps of distinct RPB's resource usage during the evaluation in Section §6.2.4. Figure 18 and Figure 19 show the results. The X-axis refers to the segments of the deployment epochs. Each segment includes 100 epochs and we discard the last epoch segment that includes less than 100 epochs. The Y-axis refers to the 22 RPBs. The color gradient of each block refers to the average memory or table entry utilization per RPB in each epoch segment. Figure 18 illustrates the allocation process of memory utilization per RPB. Our allocation scheme is a first-fit memory allocation scheme that chooses the memory to allocate when it has enough free space whatever its occupancy. In addition, memory usage of the 15 programs is also non-uniform. Thus the results of final memory allocation are also non-uniform. Figure 19 illustrates the process in which the ingress RPBs' table entries are gradually used up. The results show that the objective function $f(\mathbf{x}) = x_L$ and the hierarchical optimization perform worst as they use almost exclusively ingress RPBs. The objective function $f(\mathbf{x}) = \frac{x_L}{x_1}$ has the most uniform distribution, thus achieving the highest resource utilization and program capacity. However, the non-linear objective function brings too much computing time to the deployment delay. The objective function $f(\mathbf{x}) = \alpha x_L - \beta x_1$ has the most comprehensive performance that balances the allocation delay, resource utilization, and program capacity. In terms of the parameters α and β, we conducted a pre-experiment of parameter sweep analysis with the step size of 0.1 and the constraint $\alpha + \beta = 1$. Through this we get the best parameter $\alpha = 0.7$ and $\beta = 0.3$.