Memory Management in ActiveRMT: Towards Runtime-programmable Switches

Rajdeep Das
r4das@ucsd.edu
UC San Diego

Alex C. Snoeren
snoeren@cs.ucsd.edu
UC San Diego

ABSTRACT

A wide variety of in-network services have been developed for RMT-based switching hardware, almost exclusively through the P4 language and ecosystem. Many of these applications maintain state in switch memory, a scarce shared resource. As with any other network resource, varying traffic demands necessitate reallocations, yet the P4 ecosystem is not well suited for dynamic resource management: Modifying the set of services deployed on a switch using P4 requires the network operator to prepare a new binary image and re-provision the switch, disrupting all existing traffic. We present an alternate approach—using techniques from capsule-based active networking—to programming RMT devices that enables non-disruptive (re)allocation of switch memory at time scales that are much faster than P4 compilation without operator intervention. We use P4 to implement a single, shared runtime on commodity RMT hardware that interprets instructions received via the switch data plane to deliver a variety of exemplar services including caching, load balancing, and network telemetry. Our prototype implementation is able to dynamically provision dozens-to-hundreds of instances of simultaneous stateful services at the timescale of seconds.

CCS CONCEPTS

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

KEYWORDS

Active networking, Network Function Virtualization, P4, RMT

ACM Reference Format:
Rajdeep Das and Alex C. Snoeren. 2023. Memory Management in ActiveRMT: Towards Runtime-programmable Switches. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3603269.3604864

1 INTRODUCTION

Despite the availability of programmable switch hardware from a variety of vendors [1, 10, 44], there is considerable uncertainty regarding the viability of operator-managed service deployment. The P4 [8] ecosystem has enabled the development of a vast spectrum [22] of in-network services that can be compiled and installed on programmable switches such as those based on the reconfigurable match table (RMT) model [9]. Yet, when P4 is employed on currently available ASICs, it is not possible to alter the set of services deployed on a switch—or reallocate resources among them—without operator intervention. While Intel Tofino-based switches [1] can be re-provisioned with relatively brief (O(50-ms)) impact to traffic forwarding [5], each potential service combination must be developed and compiled independently, a complicated and time-intensive process. As a result, the existing P4-based service-deployment model limits the potential of modern programmable-switch hardware: there is no practical way to adjust a switch's service set without deep operator involvement, dramatically limiting the utility of hardware programmability in many networks and undermining the value proposition [11, 21].

In most environments, switches are shared resources, and what is needed is a way to alter the set of services (i.e., multi-programmability) and the resources allocated to each at runtime, without disrupting traffic forwarding or the functionality of existing services. Prior research on runtime RMT reconfiguration has pushed in three distinct directions: hardware extensions to support hit-less reprogramming [5, 17, 37, 41], software virtualization to enable multi-programming [20, 45, 46], and dynamic resource allocation among a fixed set of services [23, 47]. While promising, the limitations and overheads (e.g., additional crossbars) of novel hardware-based approaches are not yet fully understood—nor is any such device commercially available. Virtualization, on the other hand, has been demonstrated on commodity hardware. However, practical use cases for runtime multi-programming also require dynamic resource management which existing virtualization systems do not provide. Conversely, published approaches to dynamic memory management do not support runtime programmability.

We present an alternative approach to service deployment that enables existing RMT-based hardware to support multi-programmability and dynamic resource management: *ActiveRMT* [15]. Rather than target the natively supported RMT model through the P4 language, in ActiveRMT services are expressed in a custom instruction set that is interpreted by the switch's data plane. Services—in the form of code and data—are delivered by clients to the switch as in (capsule-based [38]) active networking. These programs are synthesized at the client on-demand according to resource allocations determined by the switch and communicated to the client through control packets. The switch runs a single, shared runtime (written in P4) that parses active packets, enforces memory isolation between services, and interprets program instructions in the data plane. The switch control plane handles admission control and resource allocation when services arrive and depart.

In ActiveRMT, program instructions are executed at line-rate directly on RMT stages one-by-one as the packet flows through

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604864

the switch pipeline: the order of instructions dictates the stage in which each instruction will execute. Because switch resources are stage-local, a service's program needs to be dynamically synthesized based upon its resource allocation. Conversely, the potential allocation for any service is constrained by the semantics of its program: a program that needs to, e.g., store a value on the switch after computing a function based upon both packet contents and existing switch state cannot be allocated memory in only one stage: it needs to first read the prior state and compute the new value before it can store the result. As a result memory allocation and service synthesis are symbiotic: clients express constraints regarding their desired allocation and the switch attempts to satisfy them while minimizing disruption to already admitted services. Isolation is ensured by requiring clients to re-target memory access instructions (akin to linking) as part of the synthesis process upon receipt of an allocation; the switch need only enforce protection.

Without requiring any changes to hardware, ActiveRMT is able to express services as capable as those implemented using P4—including caching [26], load balancing [3], and network telemetry [35]—while supporting multi-programming. We are able to make efficient use of switch resources by synthesizing program instantiations at each client that best fit with services already executing at the switch; our prototype is able to accommodate over a hundred concurrent services on a five-year-old switch (Section 6.1). Time to deployment for new services is comparable (Section 6.2) to published reconfiguration times for approaches using hardware extensions [37, 41], and resource overheads are much lower (Section 5) than in prior virtualization approaches [20, 45]. Accommodating new services does not disrupt network operation: only services—if any—whose allocations were adjusted to make room are affected (Section 6.3). This work does not raise any ethical issues.

2 MOTIVATION AND PRIOR WORK

Programmable switches are network resources and, hence, must be shared among their users. This problem is multi-dimensional. In the rest of this section, we cover some of the related work that falls within the space of potential solutions.

2.1 Modular Programming

A trivial approach to deploying multiple services is to manually combine them into one monolithic P4 program, but custom-crafting programs for each possible service combination is intractable. μP4 [36] supports modular program composition by presenting a homogenized logical architecture and uses match-action tables to provide generic packet-processing capabilities. P4All [23] similarly extends the P4 language with support for elasticity and modular programming: each independent program can make the most efficient use of switch resources by using elastic data structures that are designed to maximize memory utilization. P4Visor supports deploying multiple versions of the same service simultaneously for testing [46].

While these approaches enable modular composition, they do not solve the chief drawback of P4-based approaches: the time to deploy a novel combination of services—even if each individual service has been previously designed, debugged, and even deployed alongside a different set of services—remains dominated by the process of materializing a particular composition. Due to their various resource constraints, mapping non-trivial P4 programs onto RMT devices can be challenging [34], often requiring significant programmer expertise and compilation times on the order of minutes requiring sophisticated tools such as ILP solvers [27]. As a result, this custom compilation typically must be conducted by the operator with full access to and ability to modify the source code for each individual service. Chipmunk [18] presents an alternative (combinatorial) approach to finding feasible mappings in challenging circumstances, at a significant cost in terms of computation and time, further delaying service deployment.

2.2 Architectural Extensions

Even if a suitable binary were readily available, reprogramming currently available switches disrupts network processing for all traffic, regardless of whether it depends upon the (set of) services being (re)provisioned. While existing hardware requires blackout periods on the order of tens of milliseconds [5], researchers have proposed alternative architectures [37, 41] to allow incremental updates to the device without disrupting packet processing at all. This is achieved by modularizing switch hardware and heavily multiplexing resources across the switch using crossbars. For example, Menshen's fully isolated packet-processing modules can be independently re-configured at runtime in less than a second [37]. Unfortunately, such extensions have not made their way into commercially available devices.

Other proposed extensions seek to improve resource efficiency. For example, dRMT [12] decouples processing from memory, allowing memory to be partitioned among match-action stages according to program requirements. Follow-on work demonstrated that device behavior could be modified without disrupting operation: FlexCore [41] extends dRMT with primitives to support partial re-configuration. By breaking down various elements of a P4 program (i.e., parsers, tables, control-flow) into hardware-mapped re-configurable units, FlexCore is able to update each of these elements at runtime with varying degrees of consistency. The In-situ Programmable Switch Architecture (IPSA) [17] is another approach to enabling incremental updates to switch configuration and decoupling processor from device memory. IPSA introduces self-contained, independently programmable units known as Templated Stage Processors (TSPs) that support non-disruptive reconfigurations. The degree of multi-programmability of the switch (i.e., number of concurrent services) in such approaches, however, is limited by the number of independent hardware units.

2.3 Virtualization

Our work is similar in spirit to prior attempts to virtualize standard RMT devices [20, 45, 47]. Like ActiveRMT, Hyper4 [20] employs a generic P4 "Persona" program that runs on the device and can be configured to provide various functionality—through table updates in Hyper4's case. Their approach works well for a restricted set of functions like network slicing, snapshotting and virtual networking, but it lacks support for stateful processing required by a large number of services. Moreover, the use of resubmission to parse packets consumes switch bandwidth, and their approach to virtualization leads to prohibitive overhead [20, 45].

Virtualizing a subset of switch resources such as stateful memory is a more tractable approach on current devices. NetVRM [47] virtualizes register memory constructs on programmable switches such as the Tofino. Memory is dynamically apportioned across a precompiled set of applications at runtime through virtual addressing. While address translation is performed at runtime on the switch, page sizes are selected from a fixed set of values determined at compile time. (This, along with a two-stage cost for address translation is a consequence of the lack of hardware support for virtualization on current devices.) In addition to the coarse-grained allocations of stages (i.e. memory cannot be allocated to applications on a per-stage basis), the virtualization overheads are also significant.

2.4 Resource Allocation

Regardless of how services are deployed, they must share limited switch resources, which requires not only a mechanism to partition them but policies to determine appropriate allocations. NetVRM attempts to determine the appropriate allocation using knowledge of utility gradients and network traffic. Determining an appropriate utility function is not always straightforward, however [19]. Take the example of a telemetry service such as the count-min sketch [13]; the width of the filter determines the accuracy of the filter whereas the depth determines the probability of error in counting. Neither of these two metrics can be evaluated at runtime—otherwise it would defeat the need for the filter; they are specified at allocation time and can only be calculated using a given width and depth. Similarly, the hit rate of an in-network cache varies based upon both memory allocation and workload mix, yet the latter is a complicated function of demand, congestion control, traffic engineering, etc. ActiveRMT adopts a first-come-first-serve approach wherein new services request resources and the switch performs admission control; services with elastic demands may have their allocation reduced as additional services arrive.

2.5 Active Networking

Our approach to service deployment harkens back to design patterns from classical active networking [7, 38, 39]. While the capabilities of currently available programmable switches allow [16, 40] for a broad range of functionality, we focus on traditional in-network services like those currently supported by P4. Others have taken similar approaches in even more restricted domains. Jeyakumar *et al.* propose active packets containing Tiny Packet Programs (TPPs) [25] of up to 20 bytes in length that can take advantage of stateful processing on RMT devices, but they focus on storing and retrieving switch attributes to support network telemetry. We recognize many challenges of active-networking style approaches remain unsolved; this paper focuses on the issue of memory allocation and we defer the others to future work.

3 ACTIVERMT

The RMT [9] architecture consists of a sequence of match-action stages comprising ALUs, stateful register memory and several other hardware units that can be configured to perform a specific set of operations. Languages such as P4 [8] and Domino [34] can be used to write programs that map to such configurations to enable desired behavior. We overlay a homogenized logical architecture (shown in Figure 1) that enables network packets to determine device behavior at runtime. To this end, we pre-configure the device (using P4) to expose a set of abstractions—in the form of instructions—that can express a range of programs. (Appendix A contains the details of our instruction set.) Programs can be attached to individual packets to trigger desired behavior when the packets traverse the switch.

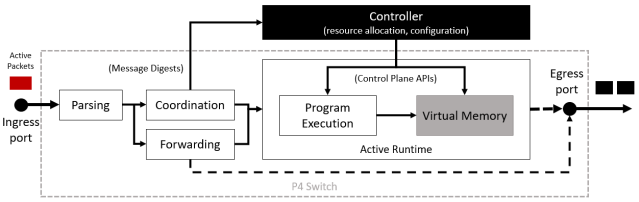


Figure 1: ActiveRMT packet processing overview

Our design allows for an essentially unlimited degree of multi-programmability as each packet executes an independent program. Behavioral and performance isolation follows from RMT's line-rate processing wherein each packet has its own independent state—contained within a packet header vector (PHV)—and does not affect the processing of other packets. Many services implement stateful processing across a set of packets (i.e., flows), however, which requires programs to access high-speed switch memory, a limited resource on such devices. As a result, in practice the degree of multi-programmability is limited by the extent to which switch memory can be shared among applications. ActiveRMT allows users to write programs that access partitioned stateful memory. Unlike prior approaches [47] we are able to dynamically partition memory—both vertically (within stages) and horizontally (across stages)—resulting in more efficient resource utilization.

3.1 Program Execution

In ActiveRMT, parsing units on a PISA switch extract code and data corresponding to an active program and store them in the PHV. Programs can have variable length and are terminated using a special `EOF` instruction. Parsed code is executed in match-action stages with one instruction being executed per physical stage. Applications that contain more instructions than the switch pipeline has stages are recirculated to continue execution. ActiveRMT defines three additional 32-bit *variables* that are maintained in the PHV: the memory address register (`MAR`) and two memory buffer registers `MBR` and `MBR2` that serve as general-purpose accumulators. Our instruction set is based on the Tofino native architecture (TNA) [4] and provides capabilities such as hashing and access to ALUs.

Instruction interpretation. Figure 2 illustrates the execution process for active programs. From a P4 perspective, the control plane installs a match table for each stage which matches on the program's FID (see Section 3.3), instruction opcode, contents of the variables, and additional control flags. Table entries define valid memory regions for each program and are computed by the control plane during allocation. We use the contents of `MAR` to enforce memory protection and the contents of `MBR` to facilitate branching. Note that both these variables can contain the results of previous operations. We implement instruction decoding using exact matches in SRAM. Memory protection is enforced through range matching in TCAMs, which end up being the resource bottleneck for the number of distinct address ranges that ActiveRMT can support.

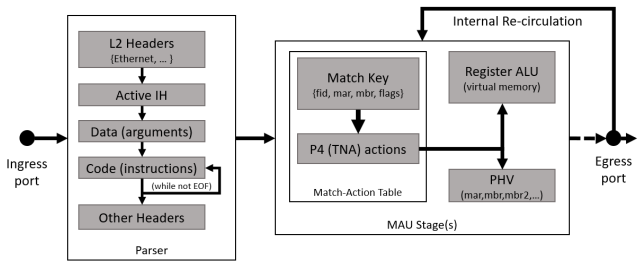


Figure 2: ActiveRMT runtime processing model

A successful match executes a corresponding P4 action which invokes a subset of primitives defined by the underlying device architecture. To support runtime programmability, we only employ primitives where the operands are all obtained from the PHV.

Control flow. Program execution proceeds sequentially through the stages of the RMT pipeline with the help of control flags. Instruction execution is enabled by default at each stage, except when there is branching or the program terminates. The latter is determined by using a control flag labeled *complete*. This flag is usually set when the `RETURN` instruction is executed. Branching occurs when a `CJUMP` instruction or one of its variants is executed. Correspondingly, a *disabled* flag is set and subsequent instructions (corresponding to the alternate branch) are skipped until this flag is reset. A branch instruction is associated with a label indicating where in the program to branch. Due to the sequential nature of program execution, this location has to be later on in the program. The flag is reset once this label is encountered.

Once an instruction is executed on a logical stage, a flag is set in the corresponding instruction header in the packet. This flag provides indication to the P4 parser that the instruction's field should be discarded from the packet. Consequently, active packets shrink in size after execution—an optimization that can be disabled if variable packet sizes are undesirable.

The full set of instructions is available in each stage, simplifying program structure. The downside of this design choice is additional overhead: A match-action stage in a typical PISA switch consists of various hardware units that can be used to implement certain programming constructs. By using match-action tables to perform instruction decoding, we are not able to take advantage of various Tofino optimizations such as checking a condition and using it to predicate table execution within the same stage. Instead, ActiveRMT typically requires conditionals to execute in a distinct stage, although there are certain instructions that can be conditionally executed without requiring an additional stage (e.g. `CRET`). Our approach similarly cannot parallelize execution within a stage.

Recirculation. There are three factors that determine whether a program can be run in one pass through the switch or requires recirculation: The first one is the program length. Programs where the number of instructions exceed the number of logical stages require packet recirculation to complete execution. (A switch can thus directly infer the recirculation cost by observing the program length.) The next one is the position (on the logical pipeline) where certain instructions are executed. For example, the return-to-sender (`RTS`) instruction should ideally be executed on a stage of the ingress pipeline, since ports cannot be changed at egress on devices such as the Tofino. (Otherwise we recirculate packets to change ports with a corresponding overhead). Finally, instructions that clone packets (e.g., `FORK`) also require recirculation.

3.2 Memory Semantics

The ability to access stateful memory from the data plane enables a large number of interesting applications. On a Tofino switch register "externs" enable this capability. Each register has its own stateful ALU for which multiple micro-programs (register actions) can be defined and selected, on a per-packet basis, from the same match table. We define memory semantics using four register ALU actions. (The resulting instructions are listed in Appendix A.4.)

Layout. In our design, we use one large register array to store memory objects in a particular stage. Based on the constraints described above, we define a set of register ALU semantics and corresponding actions which (in our experience) is enough to express a number of non-trivial applications (Appendix B). Memory is directly addressable based on the contents of the `MAR` variable and protection is enforced by the match tables. Consistent with the RMT design, a packet (and consequently an active program) can access only one memory object per stage.

Address translation. We allocate a contiguous region of each stage to a particular application. The pipeline uses physical addressing, so we need to apply a mask and add an offset to translate a program's accesses for a given allocation. Because there are no primitives on the Tofino to perform such an operation, we implement address translation as part of program synthesis at the client: the switch communicates the details of an service's memory allocation at admission, and the client updates its program's memory access instructions accordingly.

ActiveRMT can support runtime translation at the switch when necessary, however, such as to perform address translation on the result of a hash. We define instructions (Appendix A.6) to apply the appropriate mask and offset (determined by the switch at runtime based upon the stage at which the memory access will execute to ensure memory safety) to the value of `MAR`.

3.3 Program Encoding

(End-host) clients run a shim layer that is configured with the (set of) service(s) they wish to employ at the switch. Active programs *do not* operate—or even inspect—the TCP/IP payload of packets; rather, outgoing packets are encapsulated with special headers that contain the instructions[1] and data corresponding to an ActiveRMT program. The order of instructions is determined based upon the match-action units and parsers allocated to the service by the switch. As a result, the shim-layer logic for each service needs to understand the protocols used by packets upon which it operates, and encode any relevant data from the payload into the active headers.

Our prototype uses layer-2 encapsulation, following the standard Ethernet header (i.e., a special VLAN tag). While limiting its use to local networks, this choice dramatically simplifies interaction with standard transport protocols such as TCP and UDP and allows packets to be "activated" just prior to transmission in a traditional

[1]Such a design adds overhead. We could potentially optimize our solution by caching code on the switch; However, such enhancements are beyond the scope of this paper.

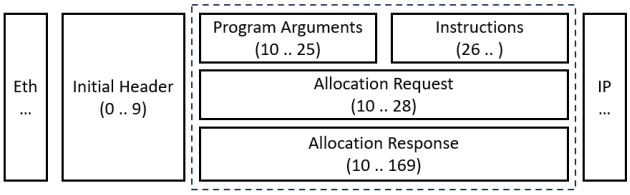


Figure 3: ActiveRMT packet header format. Numbers in parentheses indicate field byte position.

POSIX networking stack. Figure 3 illustrates the ActiveRMT header format. An *initial header* marks the beginning of an active program. This header contains an identifier called *FID* which is used to identify an active program along with control flags that determine the nature of the active packet. One of the control flags specifies the type of active packet which determines the next set of headers.

There are three types of active packets: allocation requests, allocation responses, and active programs. Allocation request packets contain a set of headers that describe an active program in terms of its memory access patterns—the length of the program, the stages where it accesses memory and the respective demands of each stage. Allocation response packets contain the start and end locations of the memory regions allocated in each stage. Active program packets comprise of a set of *argument* headers (containing program data) followed by a sequence of *instruction* headers which define the (code for the) active program.

The initial header is 10 bytes while the argument header is 16 bytes (consisting of four 32-bit data fields) followed by a variable number of instruction headers, each of which contains two bytes: a one-byte opcode and a one-byte flag. The former is used to identify the instruction to be executed while the latter is used for control flow (as described in Section 3.1). In our prototype allocation request headers are 24-bytes long, consisting of eight three-byte headers corresponding to eight potential memory accesses. Allocation response headers are 160-bytes long and consist of 20 eight-byte headers encoding the memory regions allocated in each of the 20 stages in our switch pipeline.

3.4 Example: In-Network Cache

Our instruction set allows us to implement a number of useful services. Here we present a toy example of an in-network cache that can store small objects (4-byte values with 8-byte keys) from realistic workloads [2, 42, 43] on a PISA switch. (Additional examples of active programs can be found in Appendix B.) This service consists of two separate active programs: one to store key/value objects on the switch and one to request objects if available. Listing 1 shows the active program for the latter, which clients attach to application-level read requests addressed to the server. When inserting the program into the packet, the ActiveRMT shim on the client computes a hash of the desired key (by parsing the application-layer payload of the packet) and calculates the address of the corresponding hash-table bucket on the switch based upon the memory allocation it received when registering the service at the switch (following the process described in Section 4.3).

Object retrieval. Line 1 of the program loads that address (depicted as *$ADDR*) into the `MAR` variable. In the next line, the program

```
 1  MAR_LOAD,$ADDR     // locate bucket
 2  MEM_READ           // first 4 bytes
 3  MBR_EQUALS_DATA_1  // compare bytes
 4  CRET               // partial match?
 5  MEM_READ           // next 4 bytes
 6  MBR_EQUALS_DATA_2  // compare bytes
 7  CRET               // full match?
 8  RTS                // create reply
 9  MEM_READ           // read the value
10  MBR_STORE          // write to packet
11  RETURN             // fin.
```

Listing 1: Active program for querying an object cache

reads the first four bytes of the key stored in that bucket (i.e., located at `MAR`) into `MBR`—and advances `MAR` accordingly—which is compared (line 3) with first the four bytes of the requested key stored in the first data header in the packet. If the bytes are not equal, it is a cache miss and the program terminates (line 4), causing the packet to be forwarded to its destination, presumably a server that will service the application-level request for the same object contained in the (TCP/IP) payload of the packet. Lines 5–7 repeat the exercise for the remaining four bytes of the key to check for an exact match. If the program reaches line 8, it is a cache hit and the switch is directed to return the packet back to the source. Before terminating (line 11), however, the program reads the stored value from the the last byte of the hash bucket (line 9) and writes it into the first data header (lines 9–10). When the packet is received back at the client, the shim layer can extract the value from the data header and construct an appropriate application-layer response packet. This program can be fully executed in one pass through the switch since the total number of instructions (11) is less than the number of logical stages (20) and the `RTS` instruction maps to a stage (8) within the ingress pipeline.

Data-plane cache management. Deploying a full-featured in-network cache service like NetCache [26] using ActiveRMT involves more than just the active program above. Concretely, the service needs to determine popular items and populate the cache accordingly. As a P4 program, NetCache can use the switch's match-action tables to look up keys; in other words—in the context of key-value objects—it uses content-addressable memory. The register memory that ActiveRMT exports, on the other hand, is direct (or hash-based) addressable. Moreover, match-action table entries cannot be updated via the data plane and, as such, NetCache requires a control-plane application to perform cache management.

In NetCache, the P4 program instantiates counters for stored objects in addition to sketch-based counting of every requested key while its control-plane application identifies popular items and updates the set of objects cached in the tables. Match tables can store an arbitrary set of objects so the exact set of frequent items can always be maintained. Using register memory requires a different approach: hash-based addressing like that used in our example results in collisions. Hence, the problem transforms to storing the

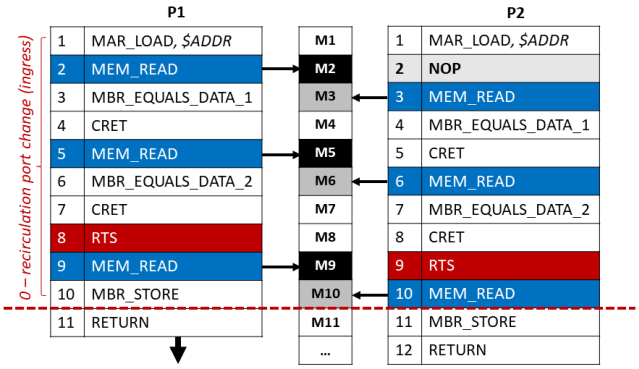


Figure 4: Mutating a program to efficiently fit within the available memory region. The red line marks the limit to which the RTS instruction can be moved.

most-frequent key-value pair among the set of keys that hash to each bucket [6, 30, 35]. Section 6.3 presents a realization of such an approach in ActiveRMT. Similarly, while NetCache relies on its control plane to populate the cache, ActiveRMT employs the data plane. Section 5 presents a set of RDMA-style client primitives that enable clients to directly access allocated switch memory. Clients of our cache service use these primitives to populate the cache in response to workload changes or due to memory reallocation.

4 DYNAMIC MEMORY ALLOCATION

Active services like in-network caching must store state at the switch. In the RMT architecture, every logical stage has its own memory which cannot be shared across stages. ActiveRMT dynamically (re)assigns memory to programs at runtime to achieve multi-programmability.

4.1 Memory Virtualization

ActiveRMT instantiates one large register array in each logical stage to be used as a dynamic memory pool. Each array fills up the entire physical memory region of its stage; the total memory available to active programs is the sum across all logical stages. At runtime, we accommodate new applications by allocating memory regions from this set of pools. Since each pool is tied to an execution stage, memory access instructions require an allocation in the corresponding stage. For the example program in Listing 1, there are three memory-access instructions: at lines 2,5 and 9. Each of these instructions require an allocation in the corresponding stages.

Because each stage is functionally equivalent, we can place any of the `MEM_READ` instructions into subsequent stages (and fill gaps with `NOP` instructions) without altering program semantics. We refer to these adjusted programs as *mutants* and exploit this flexibility when performing allocations. Figure 4 illustrates how we can mutate the cache program from Listing 1 to utilize memory in different stages. In the diagram, an instance of the cache application (P1) was allocated memory stages $M2$, $M5$, and $M9$. The allocator can avoid contention from a subsequent instance (P2) of the same application by mutating the program and inserting a `NOP` instruction at line 2 to move the memory accesses to stages $M3$, $M6$, and $M10$. As we show in Section 6.1 mutants facilitate more efficient allocation of switch resources. (Mutants that push instructions too far ahead require additional packet recirculations.)

Allocation granularity. Within a stage, the memory pool is split among currently resident programs. Unlike prior approaches [15, 47], ActiveRMT allows memory regions of arbitrary size, but analysis (Section 6.4) shows that increasing the granularity of allocation increases the time to compute an allocation. We group a contiguous set of register indices into a *block* and allocate memory at the granularity of fixed-size blocks. In our implementation, we split each stage's memory region into 256 blocks. Applications are allocated a contiguous set of blocks per logical stage. (A non-contiguous allocation can be achieved using multiple allocations.)

Elasticity. We factor in the nature of the application (characterized by its demands) when allocating memory. We defer a sophisticated utility-function-based approach [47] to future work and classify applications into two types. We refer to applications that have variable demands as "elastic" and those with fixed demands as "inelastic". An application such as the in-network cache described in Section 3.4 can be categorized as elastic: any amount of memory is beneficial, the more the better. In contrast, a stateless load balancer has inelastic demand: the amount of memory needed is based on the number of VIPs it balances among.

4.2 Allocation Algorithm

When a new service arrives, the allocations for existing applications may need to be adjusted. For existing applications, this implies moving data from the old memory region to a new one, resulting in a temporary disruption of active functionality for the affected applications. Because inelastic applications never benefit from altered (even potentially larger) allocations, we pin inelastic applications to the beginning of the memory pool in each stage. While this does not prevent fragmentation of memory when such applications depart, we speculate that inelastic applications (such as a load balancer) are unlikely to depart frequently. When computing a new allocation we consider two objectives: maximizing overall switch memory utilization and ensuring fairness among (elastic) applications.

Problem formulation. Finding a feasible allocation on programmable RMT targets is non-trivial [18, 23, 27]. Our problem, although simpler (since we only allocate memory) remains non-linear. For a given allocation, we consider the set of memory accesses resulting from all possible mutants of existing (inelastic) applications and new programs under consideration. Each candidate in the feasibility set is encoded as a fixed-length sequence of constraints on memory stage indices: a lower bound, an upper bound, and a minimum distance between consecutive memory access indices. For example, Listing 1 has $M = 3$ memory accesses at lines 2, 5 and 9, which is the most compact of all possible mutants. Thus, the lower-bound constraints are $\mathbf{LB} = \begin{bmatrix}2 & 5 & 9\end{bmatrix}$ and the minimum distances are $\mathbf{B} = \begin{bmatrix}1 & 3 & 4\end{bmatrix}$. When targeting a logical pipeline with $n = 20$ stages, the corresponding upper bounds can be computed as $\mathbf{UB} = \begin{bmatrix}11 & 14 & 18\end{bmatrix}$. Moreover, if we seek to avoid recirculation by restricting `RTS` to the ingress pipeline (i.e., in the first 10 stages), the upper bound becomes $\begin{bmatrix}4 & 7 & 11\end{bmatrix}$.

We formulate the problem of finding an allocation vector $\mathbf{x} \in \{1 \dots n\}^M$ as follows:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = g(\mathbf{x}) \cdot \mathbf{C} \\ \text{subject to} \quad & \mathbf{LB} \leq \mathbf{x} \leq \mathbf{UB} \\ & A\mathbf{x} \geq \mathbf{B} \\ \text{where} \quad & A \in \{-1, 0, 1\}^{M \times M} \\ & A_{i,j} = \{1, i = j; -1, j = i - 1; 0 \text{ otherwise}\} \end{aligned}$$

$g(\mathbf{x})$ maps from $\mathbf{x}$ to an indicator vector $\mathbf{y} \in \{0, 1\}^n$, where $y_i = 1$ if $i \in \mathbf{x}$, and $\mathbf{C}$ is a cost vector that represents the current allocation on the device. Because the objective function is non-linear we cannot use standard (I)LP solvers. Fortunately, our online allocation mechanism does not consider relocating existing applications across stages (i.e. it does not consider their mutants). Hence, a systematic search over the feasibility region can be performed in polynomial time, $O(k)$ where k is the number of mutants. Section 6.1 shows we can find solutions for our example applications rapidly in practice.

Allocation scheme. We compute the cost $\mathbf{C}$ of allocating to a particular stage based on how much "fungible" memory is available in each stage: in addition to free memory available in a logical stage, memory previously assigned to elastic applications can be reallocated to other applications. Based on this metric (and consistent with terminology in the memory-allocation literature), we refer to an allocation scheme as "worst-fit" if the scheme chooses stages that have the greatest amount of fungible memory and "best-fit" if it does the opposite. A corresponding "first-fit" approach greedily selects the first available memory region in the systematic enumeration sequence. Our prototype uses a worst-fit allocation scheme to maximize utilization; we evaluate other approaches in Section 6.4.

Fairness. Since elastic applications fill up the memory pool within a stage, they always maximize utilization within a stage. Splitting a memory pool among co-located applications within a stage raises the question of fairness, however. We follow approaches from classical network resource allocation [14, 28, 29, 31, 32] and attempt to deliver max-min fairness. Because memory is not arbitrarily divisible, we approximate it using progressive filling [32].

4.3 Allocation Process

Clients initiate an allocation with allocation-request packets (described in Section 3.3). Each such request encodes the constraints described above, effectively characterizing the application's memory access patterns. When a switch receives such a request, it communicates the information encoded in the packet to the switch controller running on the switch CPU (using Tofino message digests in our implementation). The controller serializes requests to ensure applications are admitted one at a time. The allocation is computed based on the current occupancy of the switch and the constraints specified in the request; details of a successful allocation (or failure notification) are returned to the requesting client in an allocation response packet. Once a client receives a response, it is ready to start transmitting activated packets. Section 6.2 shows the entire process takes on the order of a second.

Reallocation. During an allocation process, an existing application may be required to yield some of its previously allocated memory to an incoming application—or relocate to a different memory region entirely. We require each service to implement its own reallocation handler, which may be a simple initialization process, a straightforward copy, or an involved aggregation computation. To facilitate the task, ActiveRMT provides a consistent memory snapshot. Once a new allocation has been identified, the switch notifies the impacted applications and "deactivates" their packet programs (recognized by FID) for the duration of the reallocation process to avoid inconsistency. Once a client has completed extracting any state it wishes to save it notifies the switch using special packets containing only the global active header. Unresponsive applications are timed out to prevent them from obstructing new allocations.

State extraction. ActiveRMT provides two methods for a client to extract existing memory contents from the snapshot before its new allocation is applied and packets "reactivated." One is via the control plane (using APIs to access register memory) and the other is via the data plane (using active packets); we expect most applications to use the latter. Performing state extraction via the data plane involves writing active programs that access locations in the allocated memory region. However, since we can only access one register index per stage in the data plane, extracting a full snapshot requires sending multiple packets to retrieve a range of memory indices; retrieving contents of the entire memory region in our implementation would require 94K × 20 packets—a modest amount of traffic at 100 Gbps. Even still, to speed up the process, ActiveRMT provides primitives to read (and write to) a set of memory indices (corresponding to a set of stages) at once. The client can ensure success of the writes by programming each packet to reply back after a write through the `RTS` instruction. Packets that fail execution (i.e., are dropped) do not generate a response. Since reads and writes are idempotent the client can safely retransmit after a timeout. Appendix C shows examples of how to use these instructions; we employ them to populate the cache in Section 6.3.

5 IMPLEMENTATION

The ActiveRMT runtime consists of ≈10K lines of P4 code targeting a Tofino switch. Our controller is written in Python and comprises ≈1.2K lines of code; we use BFRT Python APIs to interact with the Tofino ASIC. Client-side support to inject and coordinate active programs is implemented in ≈3K lines of C using DPDK and VirtIO.

Switch runtime. Our P4-based runtime consumes 100% of SRAM available for register memory in each stage of our switch. We also use all of the TCAMs in each execution stage to decode instructions and enforce memory protection. That said, a full 83% of the match-action stage resources are available for active program execution. For context, in the case of a trivial cache application that reads a key and subsequently a value based on the key, even a native P4 program cannot make full use of memory in the first and last stages of the physical pipeline due to read-after-read dependencies, leading to a roughly 92% resource availability. NetVRM constrains the total addressable memory region per stage to be a power of two; as a result the overhead of its virtual address translation means less than half of the match-action stage resources are available to application programs [47].

Client compiler. An active program such as the one described in Section 3.4 has to be compiled to a set of bytes that can be inserted into active packets. In addition to generating the byte code, our compiler for ActiveRMT computes the memory access indices and ingress constraints (such as those for RTS) which are required to request allocations. It also synthesizes the appropriate mutant in response to allocation responses from the switch and performs any necessary address translation.

Shim layer. Our prototype exports a VirtIO-based Unix network socket to encapsulate and decapsulate active packets. Packets corresponding to supported active services (arriving on the virtual interface) are identified by their destination ports and parsed appropriately. Active packets arriving on the physical interface are identified by their active headers and processed accordingly.

We use a state-machine model to keep track of what state a given service and its constituent programs are in: this could be an operational state (when active programs are injected into packets being sent over the wire), a negotiating state (when an allocation is being requested/released) or a memory-management state (when state extraction is being performed). Active transmissions are paused when the client is negotiating or responding to a memory reallocation. Communications with the controller involve a poll-based mechanism with intervals around 100 μs (which is faster than the fastest allocation time).

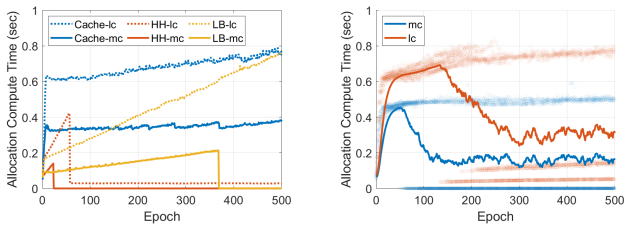
6 EVALUATION

We evaluate the performance of our ActiveRMT prototype along two distinct dimensions: 1) the effectiveness of its online memory allocator, including the impact of (re)allocations on existing services, and 2) the speed with which we can provision new services. We also include a case study of a full-featured in-network cache service as well as a comparison of different memory allocation algorithms. All of our experiments are conducted on a 64-bit, 4-core Wedge100BF-65X switch built around a Tofino ASIC connected to 20-core Intel client machines equipped with 128 GB of RAM using 40-Gbps NVIDIA Mellanox ConnectX-3 Ethernet cards. We allocate switch memory at a granularity of 1-KB blocks unless specified otherwise.

6.1 Memory Allocation

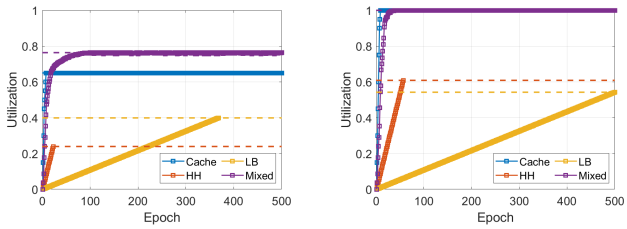
We evaluate the performance of our memory allocator when faced with different mixes of three active applications: an in-network cache [26] (as in Listing 1), stateless load balancer [3], and heavy-hitter detector [35] (with implementations in Appendix B). The cache application has elastic memory demand, while the load balancer and heavy hitter have inelastic demands of 2 blocks (enough to manage 512 active virtual IPs) and 16 blocks (to achieve less than 0.1% error with high probability) each.

Computation time. We start by ensuring that the computational task of calculating allocations is sufficiently modest so as to be performed by a switch's control processor. (For now we focus exclusively on control-plane operations and return to the data-plane aspects in subsequent experiments.) We consider two different allocation policies: one that considers only mutants that avoid additional recirculations (*most constrained*) and one that enjoys maximum flexibility at the cost of additional passes through the switch (*least constrained*). Figure 5a plots the control-plane allocation time for a sequence of 500 arrivals of cache, heavy-hitter (HH) and load-balancer (LB) application instances. We term each arrival event an "epoch". Allocation time comprises the time to search for a feasible allocation that admits the newly arrived instance plus the time to compute the final assignments for all (re)allocated instances. The latter stage takes most of the time, so epochs with failed allocations are quite brief (O(10 ms)).



(a) Pure workloads **(b) Mixed**

Figure 5: Control-plane allocation time for two different policies with workloads containing (a) a single type of application and (b) a mixture of cache, heavy hitter, and load balancer.

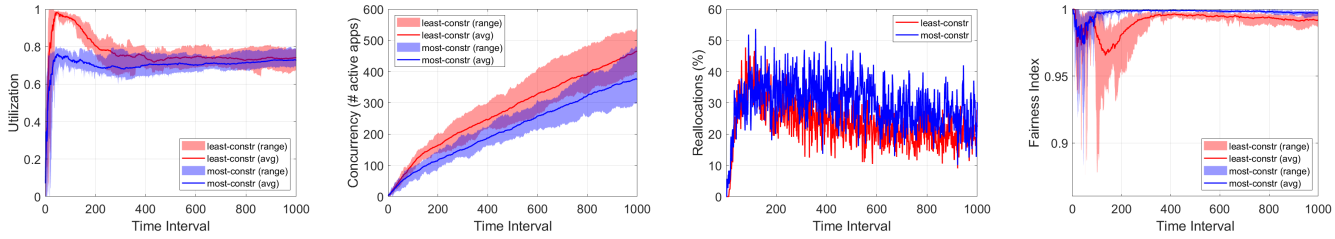


(a) Most constrained **(b) Least constrained**

Figure 6: Memory utilization of four different workloads using (a) most- and (b) least-constrained allocation policies.

As a result, the observed time collapses with the onset of placement failures. While the switch can accommodate several hundred (elastic) cache instances, inelastic applications exhaust available resources much earlier: after 23 (most-constrained) and 57 (least-constrained) instances in the case of heavy hitter and 368 load-balancer instances with the most-constrained policy. Allocations that allow recirculations to make the most efficient use of memory, i.e., those computed by least constrained (lc), take more time. The least-constrained policy considers 915, 587 and 1149 mutants of the cache, heavy-hitter, and load-balancer applications, respectively, compared to 34, 1 and 5 mutants in the most-constrained case.

Pure workloads are unlikely in practice, however. Figure 5b shows the computation time for a mixed workload (where instances are chosen uniformly at random among cache, heavy-hitter and load-balancer applications). We plot the allocation time for each arrival for 10 random trials as a scatter plot. In addition, we plot the average time across all 10 trials in each corresponding epoch as an exponentially weighted moving average (EWMA) with α=0.1 (solid lines). After around 50–150 arrivals (depending on the allocation policy) the switch cannot accommodate further inelastic applications and the only remaining successful placements correspond to cache instances. In sum, while certain cases (e.g. purely elastic applications or a long run of exclusively load-balancer instances)



(a) Utilization **(b) Concurrency** **(c) Reallocation % (elastic)** **(d) Fairness (elastic)**

Figure 7: Online allocation sequence consisting of applications randomly drawn among cache, heavy hitter and load balancer. Arrivals and departures follow a Poisson distribution with arrival rates twice that of departures.

can approach a second to allocate, arrivals from practical workloads are likely to be placed in well under a second regardless of policy.

Utilization. Figure 6 shows the memory utilization of allocations using both policies—as a fraction of total available switch register memory—plotted as a sequence of up to 500 application arrivals. While the pure cache application workload reaches its maximum memory utilization with as few as 8 instances (9 for least constrained)—at that point, there are enough different mutants to place memory allocations in all the pipeline stages its mutants can access—it can continue to admit all 500 instances. In contrast, the workload consisting entirely of load-balancer instances does not reach its maximum utilization until the allocator places 100s of instances. At that point, however, no further instances can be accommodated. Regardless of allocation policy, the maximum possible memory utilization depends on the application mix: a single application is limited in the number of stages it can utilize by its mutant set. The cache has mutants that can—at least in a least-constrained setting—make use of memory in all switch stages; the same is not true for the other two applications.

In practice, active services will arrive and depart. In order to evaluate our allocator in a realistic scenario we generate a sequence of application arrivals and departures over 1,000 unit-less time epochs. In each epoch, we draw a number of application arrivals at random following a Poisson distribution with mean 2 and departure events from a Poisson distribution with mean 1, resulting in increasing application population over time. Each new application instance is one of cache, heavy hitter, or load balancer with equal probability, while departures are selected uniformly at random from the set of currently resident applications. Figure 7a shows the utilization at the completion of each epoch for the two different allocation policies; we plot the mean across 10 trials; the shaded region depicts the range between minimum and maximum. We see that while the least-constrained policy is able to achieve a higher level of utilization initially, they both converge to about 75%.

Degree of concurrency. Figure 7b plots the number of resident applications in each epoch. As expected, the overall population grows over time, with the least-constrained policy able to place more instances due to its increased flexibility. Not all arrivals can be placed; after about 100 resident instances the allocator can only satisfy slightly less than half of the arrival instances. We expect, however, that practical scenarios are unlikely to require more than a few tens of applications to be resident at a time on any given switch. Moreover, ActiveRMT's virtualization enables it to accommodate an order of magnitude more applications than the non-virtualized alternative. For example, a minimal cache application reads a key from memory, compares it to a value in the packet, and subsequently reads the corresponding object. This application requires two memory stages: one for the key and the other for the value. When composed into a single monolithic P4 program, we can accommodate only 22 (isolated) applications (across both ingress and egress pipelines) on our switch. ActiveRMT is able to multiplex each stage across multiple instances of each mutant, supporting up to 94K instances of each mutant *in theory*. Admittedly, the utility of instances with such miniscule memory allocations is dubious.

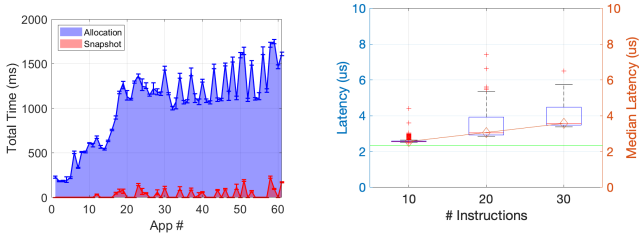
Reallocation cost. While inelastic applications are unperturbed by the arrival of new applications, elastic applications are likely to be reallocated with some frequency. Figure 7c shows the fraction of resident cache applications that are reallocated (alternatively, the expectation that any given instance will be reallocated) in each epoch in the same set of arrival sequences under both policies. To enhance visibility we plot the exponential moving average (α = 0.6). The frequency increases initially but stabilizes after there are multiple cache mutants inhabiting each stage.

Fairness. Figure 7d plots Jain's fairness index [24] among the set of cache application instances at each epoch in the online sequence. Fairness dips initially as the allocator attempts to fill up as much of the device memory as possible. Once enough cache instances have arrived, however—recall that only a third of the resident applications are cache instances in expectation—allocations converge to fair shares with the variance hovering above 0.99 in the most-constrained policy and only slightly lower for least constrained.

6.2 Latency Overhead

The time spent computing an allocation is only a fraction of the time required to actually provision a new service: the switch must update its tables and clients need to perform snapshotting on their respective memory regions.

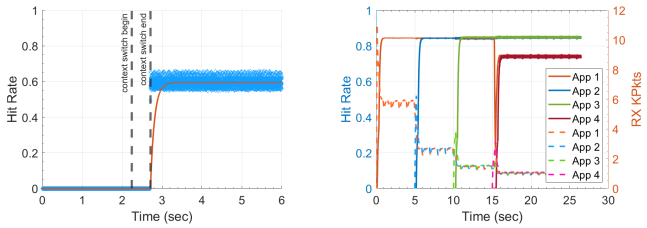
Provisioning time. Figure 8a shows the total provisioning time for each application—including any required reallocations of existing applications—for a sequence of applications arriving and departing according to a Poisson distribution as before. Provisioning time initially grows as an increasing number of existing elastic applications must be reallocated. After almost all memory is being used by some application, reallocation overhead stabilizes and the allocation time levels off at slightly over a second.



(a) Provisioning time for a sequence of 60 application arrivals.

(b) Impact of active program length on round-trip latency.

Figure 8: ActiveRMT latency overheads on a Tofino.



(a) Demonstration of demand-based allocation for an active in-network cache application.

(b) Hit rate (solid, left axis) and throughput (dashed, right axis) of four cache instances.

Figure 9: Full in-network cache lifecycle.

Provisioning time is dominated by the time taken to update table entries on the switch, including removing old entries and installing new ones based on the updated allocations. In contrast, the time required for reallocated applications to perform snapshotting ("snapshot") is a function of the number of reallocated stages and remains relatively low. This is because the total amount of memory that needs to be paged across all (reallocated) applications remains bounded by the total memory in each stage. We observe that one-to-two seconds is an order of magnitude faster than P4 compilation time on our hardware. Hence, even if we are able to (instantaneously) synthesize a composite P4 program comprising of the requisite application instances, the time to compile it would be significant. For example, on our hardware it takes it takes 28.79 seconds to compile a single P4 program for the Tofino comprising 22 instances (the maximum we can instantiate) of a semantically equivalent cache program.

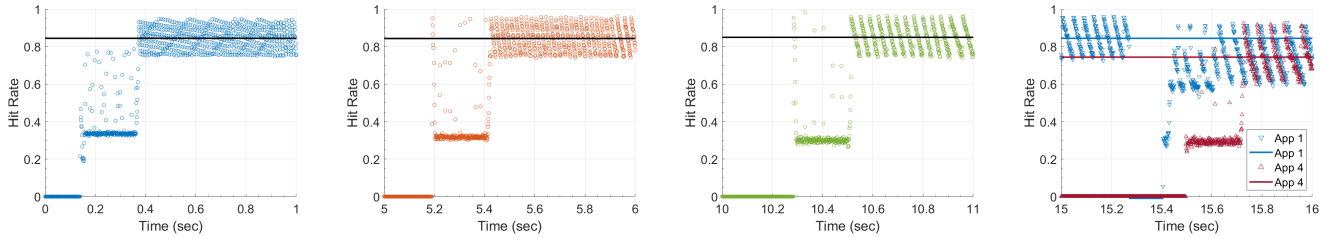
Processing latency. Once provisioned, applications execute on packets as they traverse the switch, which can introduce additional forwarding latency. To measure delay we craft active programs consisting entirely of a varying number of `NOP` instructions along with a `RTS` instruction that causes the switch to respond to the sender. We inject programs containing 10, 20, and 30 instructions into 256-byte packets and plot the (client-to-switch) RTT in Figure 8b. Because these measurements include end-host processing time, we compare to a baseline (shown in green) where the switch echos responses without any (active) processing. Our switch can process 10 instructions in the ingress pipeline, while 20 instructions require a full pass through the switch; a 30-instruction program requires recirculation. Latency increases linearly with program length; each pass through a pipeline adds approximately 0.5 μs.

6.3 Case Study

Section 3.4 presents an active program that provides basic in-network caching functionality. However, a full-featured caching service (such as NetCache) requires additional functionality such as frequent-item monitoring and cache maintenance. Here we describe an approach to implementing the full service in ActiveRMT by employing a set of distinct active programs. To use the service, a client first deploys a frequent-item (i.e., heavy-hitter) monitor to compute a set of popular objects. After a suitable amount of time, it can extract the statistics computed by the heavy-hitter program and use another program to populate the switch with a set of objects determined to be frequently accessed. Once the cache is populated, the client can then begin injecting the program shown in Listing 1 on its application-level requests.

Figure 9a illustrates precisely this scenario. For the entire duration of the experiment, a client application sends UDP (application-level) object requests containing eight-byte keys drawn from a Zipf distribution [2, 42, 43] to a remote server as fast as possible; the graph plots the fraction of these requests that are instead serviced by an on-switch cache (i.e., cache hit rate) in blue, with a smoothed EWMA shown in red. At $T = 0$ seconds, the client deploys the frequent-item application (using the allocation process described in Section 4) and activates each of its object requests with this program (shown in Appendix B.1). For a particular key requested in the packet, the program essentially performs a count-min-sketch and stores the key if the count exceeds a running threshold. The program uses packet recirculation [6] to re-access the memory stage containing the threshold and subsequently update the threshold (and store the new key). After two seconds, the client performs a memory synchronization (Section 4.3) to retrieve the thresholds and their corresponding keys. The client then begins a context switch to the cache (indicated by the vertical lines), involving deallocating the frequent-item monitor and requesting an allocation for the cache; the process completes in slightly over half a second. Once notified that the allocation process has completed, the client uses a separate program to populate the cache with the set of computed frequent items, which takes the remainder of the second. At that point, the hit rate stabilizes due to the fixed request workload used in this experiment. (The program that populates the cache could be injected on arbitrary traffic—including requests—but we use separate packets in this experiment. Moreover, while there is only one client in our testbed, in practice any of these operations could be performed by any instance from a set of clients using the (shared) cache—or even the server.)

Of course, the entire goal of ActiveRMT is to enable multi-programming. Figure 9b illustrates a scenario in which four separate clients conduct the same exercise as above, each installing their own private cache instance on the same switch, staggered by five seconds. We run this experiment from a single server and using a single core each for RX and TX; hence, bandwidth is shared. For sake of brevity, we omit the frequent-item monitor in this experiment; each client populates its cache based on known request patterns. Cache population is done at (multiplicative) intervals starting from 100 ms, rapidly populating the cache on startup. We plot the perceived hit rate of each application (at a granularity of 1 ms) using an



(a) Application 1 (b) Application 2 (c) Application 3 (d) Applications 1 & 4

Figure 10: Effective hit rate for each of the four application instances in Figure 7. Solid lines indicate the stable average hit rate.

EWMA with $\alpha = 0.01$ against the left y axis; per-client throughput is plotted against the right y axis.

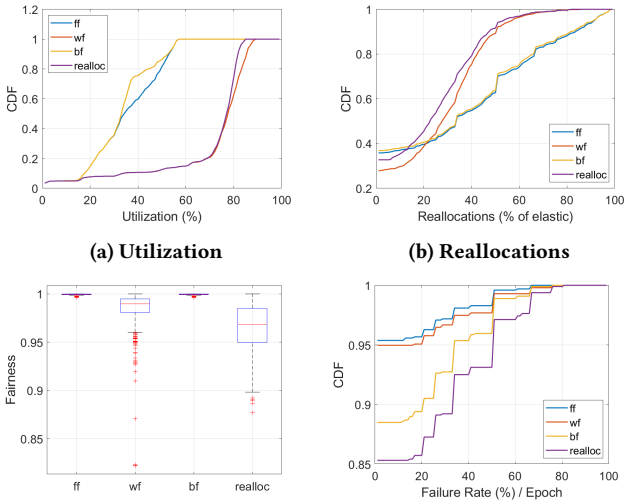
Each application instance, on arrival, triggers an allocation on the switch. The first three instances are able to take advantage of disjoint mutants (we use a most-constrained allocation policy to limit bandwidth inflation), thus obtaining exclusive memory regions (stages) and consequently zero disruption. The final instance, however, is unable to obtain an exclusive set of stages and requires sharing memory regions with the first one. Since the applications are elastic, this results in an equal—but lower—hit-rate for the two co-located instances. (Per-application) throughput is also lower than the other clients because a smaller fraction of requests are serviced by the cache as opposed to the application server.

Allocations may disrupt active functionality of concurrent applications, causing all requests to be forwarded to the server in this example. Figure 10 illustrates, at finer time scales, the perceived hit rates of each of the application instances beginning with their respective arrivals. For the first three arrivals, we observe that the hit rate starts at zero and then eventually climbs up and stabilizes at ≈85%. (The intermediate ≈30% hit rate is an artifact of the intervals used in cache population.) The duration for which each of the applications stay at zero hit-rate corresponds to provisioning time: the time taken to compute the allocations, perform snapshots, and update switch tables. Notice that when the final application arrives, the first one experiences a ≈ 150 ms disruption (at $T \approx 15.25$ seconds). During this time, the first application performs state extraction (Section 4.3) and recomputes the set of objects that need to be stored in its reduced memory region. Allocation for the incoming application then resumes and both applications perceive similar hit rates. Note that the first application can resume operation immediately after state extraction, while the incoming one has to wait for the allocation to be applied (table updates, etc.). Overall, each of the applications (irrespective of their placement) are fully functional within a second of their arrival.

6.4 Allocation Alternatives

The previous experiments all employ a worst-fit allocation policy with a fixed granularity. Here we present an analysis of alternatives.

Allocation schemes. As described in Section 4.2 we choose a program mutant and corresponding stages based on the current occupancy of the stages. We employ a "worst-fit" (wf) allocation scheme that attempts to maximize resource utilization. Here we compare with other allocation schemes including "first fit" (ff) and "best fit" (bf). The former arbitrarily chooses any feasible allocation



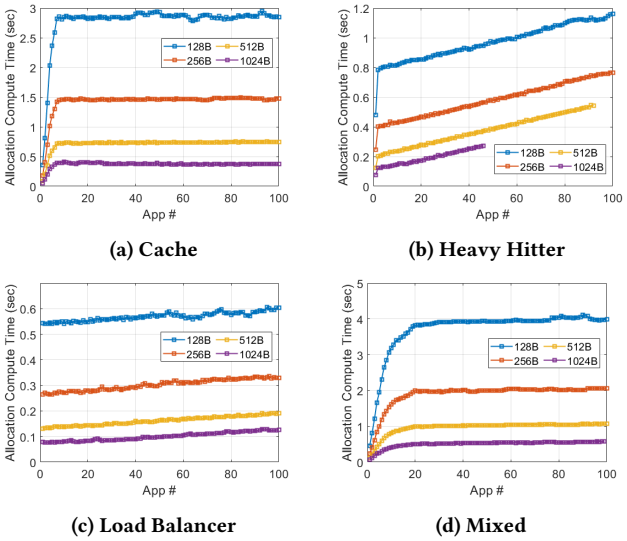
(a) Utilization (b) Reallocations

(c) Fairness (d) Failures

Figure 11: Comparison of first-fit (ff), best-fit (bf), worst-fit (wf) and reallocation-minimizing (realloc) schemes.

while the latter attempts to maximize per-stage occupancy. We also evaluate an allocation scheme that attempts to minimize the number of reallocations required to admit new applications (realloc). Figure 11 compares the allocation schemes over a simulated set of application arrivals. Consistent with our previous analyses, applications arrive for 100 epochs according to a Poisson distribution with the arrival rate twice that of the departure rate. Application instances are chosen uniformly at random from the same set of three example applications. We repeat the simulation ten times and plot the (aggregate) utilization, percentage of (elastic) applications that are reallocated, fairness index among elastic instances, and allocation failure rates across all epochs and trials. As we can see, worst fit and realloc are competitive in terms of utilization and reallocations, but worst fit has a dramatically lower failure rate. Worst-fit fairness trails that of first and best fit, but out-performs realloc and remains high in the median case.

Granularity. The amount of available memory in each stage is fixed for a particular instance of the runtime. However, the granularity with which the memory is allocated and, consequently, the number of memory blocks available in a given stage determines maximum occupancy and also impacts the time to perform allocation. The previous experiments use a granularity of 1-KB blocks. Figure 12 shows the control-plane allocation time for a sequence



(a) Cache (b) Heavy Hitter

(c) Load Balancer (d) Mixed

Figure 12: Impact of allocation granularity on control-plane allocation time for four different application mixes.

of 100 applications for four different application workloads with varying levels of allocation granularity using a most-constrained policy (c.f. Figure 5). (The switch cannot accommodate 100 heavy-hitter instances at once at 512 or 1024-B granularity.) The finer the granularity, the more complex the allocation problem becomes; the absolute impact varies across application workloads.

7 LIMITATIONS AND FUTURE WORK

While we have demonstrated the technical feasibility of supporting a large number of concurrent application instances, we have implemented only a small number of example services. As such, it is difficult to speculate on how general our initial instruction set will prove to be, or how the allocation process may handle an increased variety of demands. Moreover, practical deployment requires addressing a number of management challenges.

7.1 Limits to Generality

Internal program state is stored within PHV containers during the lifetime of a packet within the switch. Functional redundancy requires some of that state to be shared across all active program processing tables. However, due to the limited size of those containers and certain other device constraints, the total size of the shared internal state is also limited, resulting in a tradeoff between the size of memory words and the maximum number of state variables (including program arguments) that can be supported.

As mentioned earlier, our design does not allow us to take advantage of some Tofino compiler optimizations that can pack certain pieces of functionality into fewer stages. One alternative would be to use a parallel set of register variables and program-data fields, and, consequently, multiple tables. However, this may require constraining certain actions to specific stages, violating functional redundancy and potentially reducing efficiency in terms of utilization. We leave the evaluation of such alternatives to future work.

Finally, our runtime provides only baseline forwarding functionality and lacks support for many of the protocols often found on full-featured enterprise switches. Some network operators may wish to (permanently) install support for additional protocols (e.g., MPLS or IGMP). Currently, merging other P4 programs with the ActiveRMT runtime is only possible manually. For example, we integrated a subset of L2-forwarding functionality from `switch.p4`, but were forced to remove one stage from active program processing and increase the TCAM and PHV usage of the runtime by 3 and 6 percent, respectively. This extended runtime also increases latency by $\approx 4\%$. Further enhancements to baseline functionality could similarly decrease the resources available to active programs.

7.2 Deployment Considerations

Security is an obvious concern when deploying an active network. Our present model assumes a network where packets can be authenticated and dropped at the edge (e.g. using port-based access control lists). Where this is not possible, one could consider signing packets and verifying them at the switch, but the hash functions supported by our current switch are not cryptographically secure; there is no reason future hardware could not provide the necessary primitives. Even if all programs are appropriately authorized, however, recirculation provides a vector for one service to impact others in terms of available bandwidth. While ActiveRMT can impose limits on the number of recirculations, one could contemplate implementing a fairness controller that accounted for bandwidth inflation due to recirculations and rate-limited services appropriately.

In terms of inter-application isolation, RMT enforces behavioral isolation by design. For the same reasons, however, services such as firewalls cannot be implemented using capsule-based active networking, where users inject programs into packets. Options include installing a trusted host-based module that inserts active programs at the source as a form of network capability [33] or by adding a notion of privilege levels to active programs. We are exploring the latter in ongoing work.

8 CONCLUSION

We enable runtime programmability of modern RMT hardware. By leveraging capsule-based active networking, our prototype implementation is able to support dozens-to-hundreds of concurrent, stateful applications through efficient runtime provisioning of limited switch memory. Our work provides a new way to dynamically provision switch resources to deploy in-network functionality without operator intervention and at time scales much faster than the present P4-based compilation and re-configuration process. We expect, however, that practical realizations will not be used to enable arbitrary program execution by individual end users, but rather the dynamic deployment of application-specific functionality by a curated set of services.

ACKNOWLEDGEMENTS

We are indebted to G. Papen, G. Porter, V. Gurevich, and the anonymous reviewers for their comments on earlier versions of this manuscript. Our testbed was made possible through generous in-kind donations from Intel's Fast Forward Initiative and Cindy Moore's capable systems administration. This work was funded in part by the Department of Energy through grant ARPA-E DE-AR000084.

REFERENCES

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–32. https://doi.org/10.1109/HCS49909.2020.9220636

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[3] 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. 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[4] Barefoot Networks. 2021. barefootnetworks/Open-Tofino. https://github.com/barefootnetworks/Open-Tofino

[5] Antonin Bas. 2018. Leveraging Stratum and Tofino Fast Refresh for Software Upgrades. In *ONF Connect*.

[6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking* 28, 3 (June 2020), 1172–1185. https://doi.org/10.1109/TNET.2020.2982739 Conference Name: IEEE/ACM Transactions on Networking.

[7] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. 1997. Active networking and the end-to-end argument. In *Proceedings 1997 International Conference on Network Protocols*. 220–228. https://doi.org/10.1109/ICNP.1997.643717

[8] 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 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[9] 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 (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011 event-place: Hong Kong, China.

[10] Broadcom Blogs. 2020. Silicon innovations in programmable switch hardware. https://www.broadcom.com/blog/silicon-innovations-in-programmable-switch-hardware

[11] Max A. Cherney. 2023. Intel is halting development of the networking chip it got from Barefoot Networks. https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html

[12] 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 (SIGCOMM '17)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823 event-place: Los Angeles, CA, USA.

[13] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[14] Emilie Danna, Subhasree Mandal, and Arjun Singh. 2012. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*. IEEE, Orlando, FL, USA, 846–854. https://doi.org/10.1109/INFCOM.2012.6195833

[15] Rajdeep Das and Alex C. Snoeren. 2020. Enabling Active Networking on RMT Hardware. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 175–181. https://doi.org/10.1145/3422604.3425934

[16] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[17] Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. 2021. In-situ Programmable Switching using rP4: Towards Runtime Data Plane Programmability. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 69–76. https://doi.org/10.1145/3484266.3487367

[18] 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*. ACM, Virtual Event USA, 44–61. https://doi.org/10.1145/3387514.3405852

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11)*. USENIX Association, USA, 323–336.

[20] 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 (CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/2999572.2999607

[21] Linda Hardesty. 2018. Marvell Nixes the Programmable Xpliant Chip It Inherited From Cavium. https://www.sdxcentral.com/articles/news/marvell-nixes-the-programmable-xpliant-chip-it-inherited-from-cavium/2018/08/

[22] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2021. *A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research*. Technical Report arXiv:2101.10632. arXiv. http://arxiv.org/abs/2101.10632 arXiv:2101.10632 [cs] type: article.

[23] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[24] R. Jain, D. Chiu, and W. Hawe. 1998. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. https://doi.org/10.48550/arXiv.cs/9809099 arXiv:cs/9809099.

[25] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of little minions: using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2619239.2626292

[26] 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 (SOSP '17)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764 event-place: Shanghai, China.

[27] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. 103–115. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose

[28] F P Kelly, A K Maulloo, and D K H Tan. 1998. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society* 49, 3 (March 1998), 237–252. https://doi.org/10.1057/palgrave.jors.2600523

[29] Jeonghoon Mo and Jean Walrand. 2000. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking* 8, 5 (Oct. 2000), 556–567. https://doi.org/10.1109/90.879343

[30] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2022. {SketchLib}: Enabling Efficient Sketch-based Monitoring on Programmable Switches. 743–759. https://www.usenix.org/conference/nsdi22/presentation/namkung

[31] M. Pioro, P. Nilsson, E. Kubilinskas, and G. Fodor. 2003. On efficient max-min fair routing algorithms. In *Proceedings of the Eighth IEEE Symposium on Computers and Communications. ISCC 2003*. 365–372 vol.1. https://doi.org/10.1109/ISCC.2003.1214147 ISSN: 1530-1346.

[32] Bozidar Radunovic and Jean-Yves Le Boudec. 2007. A Unified Framework for Max-Min and Min-Max Fairness With Applications. *IEEE/ACM Transactions on Networking* 15, 5 (Oct. 2007), 1073–1083. https://doi.org/10.1109/TNET.2007.896231 Conference Name: IEEE/ACM Transactions on Networking.

[33] Barath Raghavan and Alex C. Snoeren. 2004. A System for Authenticated Policy-Compliant Routing. In *Proceedings of the ACM SIGCOMM Conference*. Portland, OR, 167–178.

[34] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900 event-place: Florianopolis, Brazil.

[35] 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 (SOSR '17)*. ACM, New York, NY, USA, 164–176. https://doi.org/10.1145/3050220.3063772 event-place: Santa Clara, CA, USA.

[36] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing Dataplane Programs with μP4. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3387514.3405872

[37] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. 2022. Isolation Mechanisms for {High-Speed} {Packet-Processing} Pipelines. 1289–1305. https://www.usenix.org/conference/nsdi22/presentation/wang-tao

[38] David Wetherall. 1999. Active Network Vision and Reality: Lessions from a Capsule-based System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, New York, NY, USA, 64–79. https:

//doi.org/10.1145/319151.319156 event-place: Charleston, South Carolina, USA.

[39] D.J. Wetherall, J.V. Guttag, and D.L. Tennenhouse. 1998. ANTS: a toolkit for building and dynamically deploying network protocols. In *1998 IEEE Open Architectures and Network Programming*. 117–129. https://doi.org/10.1109/OPNARC.1998.662048

[40] David Wetherall and David Tennenhouse. 2019. Retrospective on "towards an active network architecture". *ACM SIGCOMM Computer Communication Review* 49, 5 (Nov. 2019), 86–89. https://doi.org/10.1145/3371934.3371961

[41] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, and Arvind Krishnamurthy. 2022. Runtime Programmable Switches. 651–665. https://www.usenix.org/conference/nsdi22/presentation/xing

[42] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2014. Characterizing Facebook's Memcached Workload. *IEEE Internet Computing* 18, 2 (March 2014), 41–49. https://doi.org/10.1109/MIC.2013.80 Conference Name: IEEE Internet Computing.

[43] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage* 17, 3 (Aug. 2021), 17:1–17:35. https://doi.org/10.1145/3468521

[44] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 633–648. https://doi.org/10.1145/3544216.3544262

[45] 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. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 1–9. https://doi.org/10.1109/ICCCN.2017.8038396

[46] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, Heraklion Greece, 98–111. https://doi.org/10.1145/3281411.3281436

[47] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, Anirudh Sivaraman, and Xin Jin. 2022. {NetVRM}: Virtual Register Memory for Programmable Networks. 155–170. https://www.usenix.org/conference/nsdi22/presentation/zhu

Appendices

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

A INSTRUCTION SET

In this section, we describe the set of instructions that we used to write our active programs. We group our instructions into categories corresponding to data copying operations, data manipulation operations, memory access (and manipulation), control flow and special instructions.

A.1 Data Copying

Assignment instructions effectively move data between PHV containers (which contain metadata and extracted packet headers).

(1) MBR_LOAD <arg> – Loads the MBR register with the specified argument from a corresponding argument field.
(2) MBR_STORE – Stores the value of MBR into an argument field.
(3) MBR2_LOAD <arg> – Loads the MBR2 register with the specified argument from a corresponding argument field.
(4) MAR_LOAD <arg> – Loads the value of MAR with the specified argument from a corresponding argument field.
(5) COPY_MBR2_MBR – Copies the value of MBR2 into MBR.
(6) COPY_MBR_MBR2 – Copies the value of MBR into MBR2.
(7) COPY_MAR_MBR – Copies the value of MAR into MBR.
(8) COPY_MBR_MAR – Copies the value of MBR into MAR.
(9) COPY_HASHDATA_MBR – Copies the value of MBR into the hash metadata fields.
(10) COPY_HASHDATA_MBR2 – Copies the value of MBR2 into the hash metadata fields.

A.2 Data Manipulation

We enable most compiler primitives in standard P4 and Tofino with the exception of shift instructions (which cannot be virtualized).

(1) MBR_ADD_MBR2 – Performs an addition of MBR and MBR2 and stores it in MBR.
(2) MAR_ADD_MBR – Performs an addition of MBR and MAR and stores it in MAR.
(3) MAR_ADD_MBR2 – Performs an addition of MBR2 and MAR and stores it in MAR.
(4) MAR_MBR_ADD_MBR2 – Performs an addition of MBR and MBR2 and stores it in MAR.
(5) MBR_SUBTRACT_MBR2 – Subtracts the value of MBR2 from MBR and stores it in MBR.
(6) BIT_AND_MAR_MBR – Performs an AND operation between the values of MAR and MBR and stores it in MAR.
(7) BIT_OR_MBR_MBR2 – Performs and OR of MBR and MBR2 and stores it in MBR.
(8) MBR_EQUALS_MBR2 – Performs a XOR of MBR and MBR2 and stores it in MBR. This results in the value of MBR being 0 if MBR = MBR2 else a non-zero value.
(9) MAX – Computes the maximum of MBR and MBR2 and stores it in MBR.
(10) MIN – Computes the minimum of MBR and MBR2 and stores it in MBR.
(11) REVMIN – Computes the minimum of MBR and MBR2 and stores it in MBR2.
(12) SWAP_MBR_MBR2 – Swaps the contents of MBR and MBR2.
(13) MBR_NOT – Performs a bit-wise NOT operation on MBR.

A.3 Control Flow

These instructions facilitate branching and program termination.

(1) RETURN – Marks execution of the program as complete and indicates that the packet should be forwarded to the resolved destination. (There may still be additional instructions in the active packet.)
(2) CRET – Conditionally returns if true (based on value of MBR).
(3) CRETI – Conditionally returns if false (based on value of MBR).
(4) CJUMP <label> – Performs a conditional jump to the label if true (based on the value of MBR).
(5) CJUMPI <label> – Performs a conditional jump to the label if false.
(6) UJUMP <label> – Performs an unconditional jump – similar to a goto instruction certain programming languages.

A.4 Memory Access

These instructions enable reads and writes to register memory.

(1) MEM_WRITE – Writes the contents of MBR to the memory location specified by MAR.
(2) MEM_READ – Reads the contents of the memory location specified by MAR and stores it in MBR.
(3) MEM_INCREMENT – Increments the counter at the respective stage by the value of INC and stores the result into MBR.
(4) MEM_MINREAD – Reads the value of the register object and performs a min with the value of MBR.
(5) MEM_MINREADINC – Increments the value of the register object and performs a min with the value of MBR.

A.5 Packet forwarding

These instructions allow programs to impact packet forwarding.

(1) DROP – Drops the current packet.
(2) FORK – Creates a clone of the current packet and continues execution – similar to a `fork()` system call.
(3) SET_DST – Sets the destination for the current packet to the contents of MBR.
(4) RTS – Performs a return-to-sender operation. The source and destination addresses are swapped and the packet is re-directed to the source.
(5) CRTS – Performs a return-to-sender operation if condition is true (specified by MBR).

A.6 Special Instructions

Here we list a set of instructions that enable specific capabilities (similar to fixed-functions).

(1) EOF – Marks the end of the active program.
(2) NOP – Performs a no-operation – skips an instruction.
(3) ADDR_MASK – Applies the address mask for the next memory access.

(4) ADDR_OFFSET – Applies the address offset for the next memory access.
(5) HASH – Computes a hash from the values of the hash metadata fields.
(6) LOAD_5TUPLE – Loads the hash metadata fields with the 5-tuple values of the executing packet.
(7) LOAD_QDELAY – Loads MBR with the queuing delay reported by the switch.
(8) LOAD_QUEUE – Loads MBR with the queue occupancy.
(9) LOAD_PKTCOUNT – Load the current packet count at the switch.

B ACTIVE PROGRAMS

Here we list some of the active programs used in the paper.

B.1 Heavy-Hitter Detection (Cache)

Listing 2 shows the active program for computing frequent items. For our cache application described in Section 3.4 we use 8-Byte keys and 4-Byte values. Packets carry the 8-Byte value across two argument fields in the header. Lines 1 and 2 loads this value into MBR and MBR2 respectively. Lines 3 and 4 copy these values into a hashing data structure. Lines 5–13 compute the count-min-sketch update corresponding to the key. The key is hashed in line 5. The address mask and offset for logical stage 8 is applied on lines 6 and 7 respectively. On line 8, the instruction MEM_MINREADINC performs the following: a counter is incremented, the count returned is stored in MBR and the minimum of MBR and MBR2 is stored in MBR2. We do not use the minimum value now but store the value of MBR in MBR2 for use later. These steps are repeated in lines 10–13. Now, MBR2 contains the minimum and hence the sketched count. The address of the key is loaded in line 15. In line 16 we load the corresponding heavy-hitter threshold. The minimum of this threshold and the sketched count is stored in MBR in line 17. If this value equals MBR2 (line 18) then the count has not exceed the threshold and the program correspondingly terminates (line19). Since the first four bytes of the key was overwritten previously, we reload this value in line 20. We then write this part of the key to memory in line 21. We perform a trick to avoid another re-circulation by writing the updated threshold next. We first insert two *NOP* instructions to reach the memory stage for the threshold. We then copy the threshold (stored in MBR2) to MBR and write it in line 26. We interleave one of the instructions for storing the remaining part of the key in line 25. In lines 27–28 we load the remaining part of the key into MBR and write it to memory. On line 29 the program terminates.

```
1  MBR_LOAD // load key 0
2  MBR2_LOAD // load key 1
3  COPY_HASHDATA_MBR
4  COPY_HASHDATA_MBR2
5  HASH
6  ADDR_MASK
7  ADDR_OFFSET
8  MEM_MINREADINC
9  COPY_MBR2_MBR
10 HASH
11 ADDR_MASK
12 ADDR_OFFSET
13 MEM_MINREADINC
14 COPY_MBR_MBR2
15 MAR_LOAD
16 MEM_READ // read hh threshold
17 MIN
18 MBR_EQUALS_MBR2
19 CRETI
20 MBR_LOAD // reload key 0
21 MEM_WRITE
22 NOP
23 NOP
24 COPY_MBR_MBR2
25 MBR2_LOAD
26 MEM_WRITE
27 COPY_MBR_MBR2
28 MEM_WRITE
29 RETURN
```

Listing 2: Active program for computing frequent items for a cache with 8-Byte keys and 4-Byte values.

B.2 Cheetah Load Balancer

For load-balancing we adapt the P4 based approach to the Cheetah load balancer [3] into our active approach. Consistent with their implementation, there are two functions – one that selects a server for a flow and the other that routes flows to the selected server. We present active programs for both these functions. The server selection function is inserted into TCP SYN packets while the other packets carry the active program for flow routing.

B.2.1 Server selection. Listing 3 shows the active program for selecting a server. In this implementation, the VIP pool size, the VIP pool and the page table for the VIP pool is stored in memory. We use a round-robin scheduler for selecting a server and assume pool sizes to be a power of two. The program begins with loading the TCP 5-tuple into a hashing data structure in line 1. The address of the VIP pool size is then loaded (line 2) into the address variable and translated accordingly (lines 3–4). The subsequent instruction then reads the bucket size and saves it to MBR2. In line 7 the a counter is read and incremented, which is used to select the next server in a round-robin fashion. The counter value is then loaded into the address variable (MAR) and the bucket size into MBR (lines 8–9). The offset for the next server is then computed in line 10 and stored in MBR and MBR2 (lines 11–12). We then load the address for the VIP pool page table and apply the necessary translations (lines 13–15). The location of the VIP pool is read in line 16. In line 17, we apply the offset (to the server) computed earlier to the base address of the VIP pool to get the address of the server. We then read and set the corresponding port to the server (lines 18–19).

Once the server port (identifier) is obtained, we store it in a "cookie" according to the CheetahLB implementation. The server port is saved to MBR2 and MBR is loaded with a "salt" (lines 20–21). This value is loaded into the hashing data structure which contains the TCP 5-tuple (line 22). In the next line, the hash of the salt and

```
 1  LOAD_TCP_5TUPLE
 2  MAR_LOAD,$VIP_ADDR
 3  ADDR_MASK
 4  ADDR_OFFSET
 5  MEM_READ // mbr now has bucket size
 6  COPY_MBR2_MBR
 7  MEM_INCREMENT // mbr now has counter
        value
 8  COPY_MAR_MBR
 9  COPY_MBR_MBR2
10  BIT_AND_MAR_MBR
11  COPY_MBR_MAR // mbr now has round-robin
         server index
12  COPY_MBR2_MBR
13  MAR_LOAD,$VIP_ADDR
14  ADDR_MASK
15  ADDR_OFFSET
16  MEM_READ // mbr now has offset to VIP
        pool
17  MAR_MBR_ADD_MBR2 // mar now has address
         to VIP
18  MEM_READ // mbr now has VIP
19  SET_DST
20  COPY_MBR2_MBR
21  LOAD_SALT
22  COPY_HASHDATA_MBR
23  HASH
24  COPY_MBR_MAR
25  MBR_EQUALS_MBR2
26  MBR_STORE
27  RETURN
```

Listing 3: Active program for SYN packets in CheetahLB.

the 5-tuple is computed and stored in MAR. This value is copied to MBR and a bit-XOR is performed between this value and the server port (lines 24–25). We then store this value into the packet headers (line 26) and terminate the program in line 27.

B.2.2 Flow routing. Listing 4 shows the active program for routing flows on the switch based on the server selected using SYN packets. Consistent with the CheetahLB approach for stateless load balancing, we compute a hash of a (switch-specific) salt and the TCP 5-tuple and XOR it with the cookie to obtain the server port. Lines 1–2 load the 5-tuple into a hashing data structure and the salt into MBR, which is loaded into the hashing data structure in the next line. The hash is computed and stored in MAR in line 4. In line 5 the cookie is loaded from the packet headers and copied to MBR2 in line 6. We then copy the hashed value into MBR (line 7) and perform a XOR with the cookie on line 8. The result stored in MBR is then used to determine the destination port for the packet (line 9). The program returns on line 10.

```
 1  LOAD_TCP_5TUPLE
 2  LOAD_SALT
 3  COPY_HASHDATA_MBR
 4  HASH
 5  MBR_LOAD,$COOKIE
 6  COPY_MBR2_MBR
 7  COPY_MBR_MAR
 8  MBR_EQUALS_MBR2
 9  SET_DST
10  RETURN
```

Listing 4: Active program for non-SYN packets in CheetahLB.

```
1  MAR_LOAD,$ADDR
2  MEM_READ
3  MBR_STORE
4  RETURN
```

Listing 5: Remotely reading a memory location.

```
1  MAR_LOAD,$ADDR
2  MBR_LOAD,$DATA
3  MEM_WRITE
4  RETURN
```

Listing 6: Remotely writing a memory location.

C MEMORY SYNCHRONIZATION

A (re)allocation process may involve synchronizing memory regions with the client. We use activate packets containing programs to read/write memory locations to perform synchronization. We use direct addressing to access the memory locations. Here we describe these active programs.

C.1 Memory READ

Listing 5 shows the active program for reading a memory location. Note that with such a program, memory regions in the first logical stage are not accessible (due to the MAR_LOAD instruction). The active compiler performs an optimization to get around this limitation by "preloading" values (such as MAR) before active execution begins. Thus the program can be re-written in a way that omits the MAR_LOAD instruction (enabling access to the first memory region).

C.2 Memory WRITE

A corresponding memory write active program can be found in Listing 6. Notice that in this program, an additional MBR_LOAD instruction precedes the memory access instruction. Our "preloading" trick is applied here as well to MBR, allowing memory writes to every memory location in the active memory region.