UNIVERSITY OF CALIFORNIA SAN DIEGO

Multiprocessing and Runtime Programmability on Virtualized RMT Switches

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Rajdeep Das

Committee in charge:

Professor Alex C. Snoeren, Chair
Professor George Papen
Professor George Porter
Professor Geoffrey Voelker

2024

University of California San Diego

2024

DEDICATION

Dedicated to my Grandparents

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to begin by thanking my parents and my sister for supporting me in my endeavors. I would also like to thank my friends and labmates for all the arguments and intellectual discussions that further drove my passion for learning.

I would like to acknowledge Professor Alex C. Snoeren for his support as the chair of my committee and as my advisor since the beginning of my doctoral journey. Through multiple drafts over both this dissertation and related conference publications, his guidance has proved to be invaluable. I would also like to thank Professor George Porter, Professor George Papen and Professor Geoffrey Voelker for being on my doctoral dissertation committee and for their roles as teachers, mentors and coauthors on conference publications, during the course of my doctoral program.

Chapter 2 contains research that was supported in part by the National Science Foundation (through grants CNS-1564185, CNS-1629973, and CNS-1911104) and the Advanced Research Projects Agency (ARPA-E). I would like to thank the anonymous reviews for their comments on an earlier draft of the manuscript prior to publishing at ACM Hotnets 2020. I would also like to thank Vladimir Gurevich for his training on P4 and the Tofino platform and also the Barefoot FASTER community for their assistance with P4 programming.

Chapter 3 contains research that was funded in part by the Department of Energy through grant ARPA-E DE-AR000084. I am indebted to George Papen, George Porter, Vladimir Gurevich, and the anonymous reviewers for their comments on earlier versions of the manuscript prior to publishing at ACM SIGCOMM 2023. Our testbed was made possible through generous in-kind donations from Intel's Fast Forward Initiative and Cindy Moore's capable systems administration. I would also like to thank members of the Intel Connectivity Research Program for their assistance with programming.

Chapter 4 contains unpublished material that was co-authored with Alex C. Snoeren from UC San Diego. This research was funded in part by Cisco Research.

VITA

2009–2013	Bachelor of Technology, Information Technology, West Bengal University of Technology, Kolkata
2013–2015	Master of Technology, Computer Science and Engineering, Indian Institute of Technology, Kanpur
2013–2015	Teaching Assistant, Department of Computer Science and Engineering Indian Institute of Technology, Kanpur
2015–2017	Research Fellow, Microsoft Research India, Bengaluru
2017–2024	Graduate Student Researcher, Department of Computer Science and Engineering University of California San Diego
2019, 2023	Teaching Assistant, Department of Computer Science and Engineering University of California San Diego
2024	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

Multiprocessing and Runtime Programmability on Virtualized RMT Switches

by

Rajdeep Das

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Alex C. Snoeren, Chair

Reconfigurable match tables (RMT) have been widely adopted in practice over high-speed packet processing pipelines. Coupled with P4, a number of useful application-specific tasks such as in-network telemetry, key-value caching, aggregation and load balancing, have found their way into the network. However, achieving multi-tenancy on such devices has not been a trivial task. RMT switches can run only one program per processing pipeline and multi-tenancy is currently achieved using static program composition with the inability to perform runtime updates. Moreover, memory is local to processing stages making it difficult to achieve efficient resource utilization. I first present ActiveRMT, a capsule-based approach to leveraging computation within the network using a general purpose memory-efficient packet processing model that pre-configures match tables to execute user-defined programs at runtime. Using a fast coordinated approach to dynamic memory allocation along with a constraint-guided approach to synthesizing stateful active programs, I present a unique method of hitlessly provisioning computationally cheap tasks with low memory footprint, that operate on a per-packet basis, onto a programmable switch. However, a capsule-based approach limits the scope of network functionality, particularly in terms of behavioral inspection. Hence, I present vRMT, a system that expands the set of tasks that can be deployed over such a packet processing runtime to include both commonly used network functions and application offloads. I show how network functions that perform behavioral inspection on arbitrary packets – using programs defined by an authorized third-party (such as a network operator) – can co-execute with application-specific tasks using automated filter composition and function chain synthesis. Generalizing recirculation-to-completion as a technique to accommodate such function chains, I present a unique method of deploying such combinations of network functions over a best-effort programmable networking substrate. We address a key challenge to supporting complex function chains by showing how to effectively manage switch backplane bandwidth when recirculating packets through RMT pipelines.

Introduction

Programmable switches [12, 42, 91] today enable offloading application functionality onto energy-efficient and high-throughput network processing devices capable of general purpose compute, by moving functionality closer to the network. Deploying functionality in this way helps cut down on network latency and improve performance and efficiency of network functionality. Commercially available RMT devices such as Tofino can achieve port-to-port latency of less than 400 ns and packet processing throughputs of several terabits per second. Introducing programmability on such devices has enabled the rapid evolution of network functionality and has subsequently led to the research community witnessing a large number of interesting and useful applications that could be offloaded onto such network processors – such as load balancing [4, 63, 78], co-ordination [45, 59, 97], caching [46, 60] and machine-learning [74, 88] to name a few [38, 51].

With the emergence of powerful programmable switch hardware, one can scale to more network functions with more (cheap) programmable switches and reuse spare resources for other networking tasks. Conversely, excess capacity on networks can be used to run network functions. Such a proposition is not unreasonable since networks are usually overprovisioned. Essential network functions can be assigned dedicated resources on the same grounds as dedicated VMs. While network function virtualization over general purpose servers have been debated to be more practical and cost-effective over building custom middleboxes, enhancing switches with programmable capabilities now raises the debate about how and to what extent such devices can be harnessed to perform such tasks, given their presence within the network. The answer to this question is not obvious since programmable switches do have limited compute and memory, and expanding hardware comes at a cost to performance. However, one can argue that it is more reasonable to scale switch capacity instead of diverting traffic to general purpose servers, complicating traffic engineering and significantly increasing the cost of network function processing. We thus attempt to reason about whether network functions can be deployed at scale over programmable switches. Fortunately, programmable switches with enhanced hardware [12, 42, 91] are evolving to facilitate such a goal.

However, the most widely known ecosystem of RMT-based [12] programmable switches, as of today, is limited in its ability to host a multi-tenant environment that can be dynamically provisioned. Programming such devices typically involve writing a P4 program that is loaded onto the switch (requiring a device reset). A packet provides a context for invoking the program, which subsequently initiates a process that performs various operations on the packet. The devices are provisioned with redundant hardware, that allow parallel execution of the same program over multiple packets (processes). However, switches receive heterogeneous traffic requiring a multiprocessing environment to be hosted by the device. Separation of functionality among various network slices can be achieved by composing all such functionality into one monolithic program and defining respective filters within the program. Yet, such an approach requires reloading the device every time the program is changed (e.g. when migrating a function from one switch to another), advocating the need for a runtime programmable environment to enable applications to be deployed onto the switch without causing network disruption.

While several approaches to enable runtime programmability [36, 87, 93] have been proposed in the recent past (some of which have been implemented on NICs [87]) there hasn't been much on the front of programmable switches. Thus, current approaches to multi-tenancy (multi-processing) that require static program composition [27, 53, 96, 98] suffer from the same problems. To such an end, we propose approaches to virtualizing RMT-based programmable switch hardware that enables both multiprocessing and runtime programmability on programmable switches. Our first system, ActiveRMT, enables a means of invoking functionality – defined by network users – on programmable switches, using encapsulated packets that contain programs.

This allows applications to be offloaded onto network switches without requiring operator intervention. Our second system, vRMT, enables authorized parties to deploy virtual network functions that can process arbitrary packets, onto such devices, thus allowing functionality such as behavioral inspection to be deployed over virtualized RMT switches. These systems further provide a unified way to allow flexible switching hardware to be more accessible to network users, by enabling application service providers and network operators to program switch computational elements – application services could offload functionality onto high performance network processing elements within switches, while operators could implement policies over the same device.

State-of-the-Art

P4 provides an ecosystem that consolidates network programmability by allowing a general-purpose mode of expressing network functions across a large set of programmable network devices. Yet, the burden of coalescing various functions – such as standard network forwarding protocols along with application offloads – lies with the P4 programmer; P4 programs are monolithic and targeted towards one device. The task of (programmatically) implementing network forwarding functions alongside experimental (or application accelerating) functions could be fragile – incorrect implementations may break the network. One solution to this may be to separate fixed functionality from programmable functionality and disallow network forwarding functions from being programmable. A better alternative is to enable multiprocessing and runtime programmability. This would enable applications to be added and removed from the switch without affecting other applications or network operation, and also allow behavioral isolation among the applications.

RMT is however inadequate when it comes to achieving such objectives. This is where application virtualization becomes relevant. Network function virtualization today is usually implemented over x86 virtual machines which are not throughput optimized. Various frameworks have been built around such an ecosystem to allow efficient packet processing and also to homogenize such a virtual network processing system over x86 machines. Programmable packet processing devices are however throughput optimized. The challenge is then to be able to virtualize such devices for multiprocessing and enable efficient use of resources. Virtualizing such programmable network devices can also enable rapid deployment of experimental functionality along with fixed network functions. Current network function virtualization platforms are however built on top of general purpose servers (often augmented with NIC-based accelerators [20]). The model used to describe network functions and most techniques used to enable efficient processing are not applicable in the context of programmable switches. However, problems such as packet classification and chain synthesis do apply. Even then, the extremely limited switch resources present several other challenges and require additional explorations to realize their design and implementation.

There has not been much addition to P4 in terms of language and programmability, even in the context of multi-tenancy on switches. Approaches have mostly included static program composition [27, 53, 98] to accommodate functionality from a set of programs onto a single device. This has typically been enabled by templating the P4 language [53, 98]. Alternative languages that enable specific functionality (e.g. Domino [76]) have been integrated as target-specific language constructs within the P4 language. The language system has been easily expandable and semantically compatible to expressing functionality directed towards relevant device targets and runtime environments. While the functionality compatible with our systems are, in theory, expressible in some extended P4 language (or constructs) such explorations are beyond the scope of this work. Instead, the focus is on defining a higher level domain specific language that could enable programming the virtualized switch.

The mode of programming devices has also been restricted to the P4 language and associated control plane APIs (e.g. the P4 runtime). Alternative modes of program delivery such as active networks [85] have not been well explored. While a general purpose active network built on programmable switches has not existed until now, there have been attempts to embed a small set of instructions within packets [44] to perform a limited set of user-defined functionality. Using table configurations to deliver programs to switches have also been attempted [36, 93], although in a limited setting (e.g. no stateful processing) and with impractical overheads.

Multi-tenancy has further been an elusive goal. Recent attempts to virtualize switch resources such as stateful memory [39, 98] require static program composition. Programmable switches based on RMT pipelines do not have hardware support for virtualization, requiring expensive hacks and workarounds [98]. In addition, match table structures cannot be reconfigured hitlessly on current RMT switches (for instance, due to the packing algorithms used to compile programs). Thus, the lack of support for runtime programmability makes them less practical to deploy and also decreases the efficiency of such approaches.

Limitations

Several challenges are associated with achieving a multi-tenant runtime programmable environment on RMT-based programmable switches. Achieving multi-tenancy requires the separation of hardware resources based on application functionality. It additionally requires performance and resource isolation. Due to the limitations of static partitioning of resources with respect to a runtime programmable environment, we focus on techniques to provision resources virtually at runtime over RMT pipelines. We first describe some of the general challenges in achieving our objectives and then discuss how our approaches introduce additional challenges.

P4 semantics have limited support in achieving runtime programmability on current RMT devices. While prior work has attempted virtualizing P4 applications using match-tables [36, 93] the overheads of such approaches are impractical in any realistic setting. Attempting to express P4 tables on match-action pipelines requires an excessive number of processing stages to emulate both match and action operations. Moreover, due to such overheads, these approaches also require a significant number of recirculations to express programs that do not even require any stateful processing – a feature necessary to implement practical in-network functionality. We argue that attempting to port programs from the P4 language system to a virtualized environment constructed using match tables leads to an inherent semantically incompatible problem. Rather, a high-level language system should be designed that is able to express functionality with a similar degree of expressibility as P4. P4 on the other hand should be used to define low-level interfaces that enable such a programmable environment.

Runtime programmability requires the ability to modify application functionality on a programmable switch while it is still processing packets. Since current RMT devices such as Tofino require a device reset to be performed every time a program is changed, a disruptive network is inevitable with static approaches to modifying switch behavior. Even rapid reconfiguration approaches require tens of milliseconds, which in the context of terabit-scale programmable switches results in dropping gigabytes of traffic. While there have been recent approaches [87] designated for other classes of network processors (e.g. NICs [20]), current runtime programmable approaches also have their limitations, for example, requiring the use of stashes (or auxiliary memory) which may be unavailable on-chip. Exploring such a direction requires considering how to stash memory objects when performing hitless updates.

While performing updates to functionality may be ultimately desired, certain use cases may suffice with dynamically allocating resources such as memory among co-located applications on a switch. The design of RMT pipelines are inherently inefficient in memory usage – dependencies could lead to unused memory in earlier stages. A solution to such problems have been proposed through alternative hardware designs [15]. However, such architectures have not been realized on programmable switches and their feasibility is also unknown. There have however been attempts to enable dynamic memory allocation [98] over RMT switches using virtualization, although at a significant resource overhead – existing approaches is able to dynamically reallocate around 50% of switch memory – the overheads of isolation (including address translation) are significant. The challenge is to come up with a technique that allows dynamic remapping of memory on RMT switches with a low overhead.

Thesis

Using virtualization, it is possible to achieve efficient multiprocessing and runtime programmability over a software defined active network consisting of RMT-based programmable switches.

We begin in Chapter 1, by describing tools, techniques and systems that enable programmable packet processing over both conventional general purpose processors as well as custom hardware accelerators. This also includes language systems and associated frameworks. Based on existing literature, we then describe how techniques such as modularization, virtualization and resource sharing have been realized on such ecosystems.

Chapter 2 describes our ActiveRMT system which presents a packet processing model that can be used to deploy an active network using capsules over RMT-based programmable switches. We describe the programming model along with associated set of instructions and demonstrate using examples how such a programming model could be used to offload application functionality. We correspondingly describe the memory model and how switch memory is virtualized. We then describe memory access semantics and its utility. Our initial exploration focuses exclusively on determining whether it is worthwhile from a performance perspective to reconsider active networking given today's hardware capabilities. Many challenges to running arbitrary code embedded in network packets remain, most notably the serious security implications. Previous proposals such as signing the active program and using the control plane to make the network aware of accepted signatures could still apply, but there have been recent advances on this front as well. Moreover, the fixed-function capabilities (such as hashing) offered by programmable switches may help in restricting the hazards of untrusted code.

Our next exploration considers stateful processing to enable meaningful applications to be implemented using such a framework. Most notably is the challenge of multi-tenancy, where multiple applications are expected to share pooled resources such as memory. While there have been recent proposals towards this goal using statically composed programs, a dynamic environment presents both challenges and opportunities for efficient resource management. The question arises as to whether efficient memory management can be achieved using virtualized switch memory. Chapter 3 describes the memory management techniques used in our approach. We describe how virtualized memory is dynamically allocated using a coordinated mechanism. We evaluate such a mechanism with respect to resource efficiency and discuss associated overheads.

Our final exploration attempts to answer whether such a system can be adapted to offload commonly used network functions onto programmable switches, especially ones that require behavioral inspection. Realizing protection domains, packet classification and authentication form auxillary problems that require exploration to deem feasibility and assert practicality. Understanding the network costs behind such a packet processing model (e.g. bandwidth), is explored as a further step. How such a model affects the fairness and efficiency of network resources using traditional networking protocols is investigated. Chapter 4 correspondingly describes the vRMT system which enables application-defined programs to perform behavioral inspection over network packets and co-exist with application offloads over a virtualized function execution environment. We focus on one critical aspect of such a system, function chaining, and how it is realized over programmable switch hardware. We evaluate the practicality of the approach in terms of degree of multi-processing and switch bandwidth utilization.

We finally conclude and discuss the implications of our approach and future considerations in Chapter 5.

Challenges

Our approach entails a virtualized platform over programmable switches that enables multi-processing and runtime programmability. There are several associated challenges here. First, the mode of program delivery commmands exploration. We use active networking to such an end. Programs specified by network packets can further accelerate experimental networking by allowing applications to define network behavior. Hierarchical layering of experimental network protocols can be replaced by a generic active protocol which allows application service providers to define network behavior over the internet protocol. In case one communication mechanism proves to be successful (in some regards) over time, it can be adopted widely; in other words this enables incremental deployment of an experimental network. This also allows deployment of short-lived packet-processing tasks (including application offloads) over the network.

In addition to security and performance implications, there are several other pragmatisms associated with deploying an active network. One such consideration is network goodput. Since active programs can occupy a significant fraction of a network packet payload, this effectively reduces the goodput of the network. This payload does not contain information that is consumed by the end-hosts and is hence an overhead to the network. One way of solving this problem would be to cache active programs on the switches to the extent possible. Similar approaches were proposed previously [84] where a function identifier could be used to retrieve code stored on the forwarding device; the capabilities of current programmable switches can further optimize such an approach.

Resource management policies are also a concern. Applications such as caches will benefit less from dynamic reallocation of resources than applications whose performance scales linearly with memory such as load balancers. Policies need to be weighted according to the impact they have on application and network performance. For example, longer active programs that require multiple recirculations (such as our object cache) consume more switch-plane capacity. While active network programmers need to keep this in mind while writing programs, some assistance from the network may help minimize the potential negative impacts on both network as well as application performance.

Keeping such things in mind, we believe that the large emerging space of application functions that are being built on top of the P4 ecosystem makes it worth reconsidering the way these functions are executed on a P4 switch. It may make sense for network behavior to be controlled by end-host applications without having to interact with network administrators. Recent approaches like INT [52] aim to achieve this for specific use cases such as telemetry. Further, service chaining in virtual network functions also seems likely to benefit from end-host control. Our approach attempts to deliver the flexibility to meet all of these requirements using ideas from active networking introduced decades ago, but applied to modern programmable switch hardware. As recently pointed out [85] by researchers who introduced active networking, the ecosystem built around programmable switches has enabled achieving something that was frequently perceived as lacking utility earlier.

Chapter 1

Background

Network functionality such as load balancers and firewalls have been implemented over a variety of hardware – ranging from general purpose (x86) processors to application-specific integrated circuits (ASICs). The former allows the most amount of flexibility, enabling arbitrarily complex applications to be deployed over the network data path. The latter achieves the most efficient form of network processing, allowing functionality designated for the network to be pushed to high-radix, high-throughput line-rate forwarding devices such as switches and routers. Yet, until recently most of such hardware has had fixed functionality baked in at the time of fabrication. Programmable switches balance flexibility with performance, enabling high performance evolving networks. Coupled with an intuitive programming model, this ecosystem has witnessed significant popularity in the recent past, particularly in the context of application offloading. During the same time, network function virtualization over general purpose CPUs (often coupled with accelerators) have also witnessed several advancements.

This chapter covers some background and related work on hardware, tools and techniques related to network packet processing. We begin in Section 1.1 by describing some of the models used in network packet processing and how they have been adopted in various hardware and software frameworks. Then, in Section 1.2, we describe network function virtualization – a widely used technique for packet processing over general purpose CPUs – and its associated frameworks. Section 1.3 describes the class of hardware accelerated programmable packet processors that have been integrated into network switches. We describe how such ecosystems have evolved over the recent years in terms of programmability and resource management. In Section 1.4, we then move on to language systems associated with such hardware and throw some light on how they have evolved over time. Subsequently, in Section 1.5, we briefly describe the space of active networking – a technique that we have incorporated into our systems.

1.1 Packet Processing Models

There are various modes of performing network packet processing today, on devices ranging from general purpose CPUs to application specific integrated circuits (ASICs). The emergence of network function virtualization has led to the development of highly efficient packet processing frameworks over the former, while several use cases favor the latter due to their high performance and efficiency.

Some of the earlier approaches to modelling a packet processor dates back to the 1990s, with the introduction of the Click modular router [66]. In this approach, a processor is composed of a set of modules or “elements" that can be chained together using interfaces. Each interface defines semantics that enable transfer of control to another element. The elements are self-contained in their behavior – functionality determining how to process the packet, what to do with it, and how and what state to maintain is contained within an element. Mechanisms such as “push", “pull" and “queue" stitch together a packet processing pipeline. Click is one of the earliest examples of a software router that can be run over general purpose Linux systems. Later on, various adaptations of Click have emerged [3, 62] that make use of efficient packet processing over optimized frameworks.

Today, a packet processor typically consists of a parser, transformer and deparser. The parser extracts packet headers using a state machine, the transformer performs various operations on the parsed packet – such as mangling, forwarding, aggregating – and the deparser reconstructs the processed packet. Such a model allows for the construction of efficient pipelines that can be implemented in software or hardware – parsers and transformers can be implemented (using algorithms and architectures) to run at line-rate in commercial settings. Thus, fixed-function ASICs and software routers alike, perform packet processing based on such a model.

1.1.1 P4

The P4 [11] language provides a programming model for both hardware and software implementations of such packet processors. A pipeline typically consists of a parser, a pre-processor, a traffic manager, a post-processor and a deparser. The implementation of these units is target-dependent; The abstractions are not (P4 as a language is not truly target-independent). The parser extracts and names headers to be used in subsequent phases. The pre-processor performs a set of operations prior to queueing and switching, which opens up the opportunities to program such behavior. The traffic manager performs the queueing and switching but is usually treated as a black box (implemented by the respective targets) in the model. The post-processor opens up more opportunities such as collecting statistics or mangling packet headers. Finally, the deparser performs standard network operations (such as checksum computation) and reconstructs the packet. This model closely maps to a range of targets, most notably the protocol independent switch architecture (PISA).

These models give us an insight into the design of packet processing pipelines. The argument is that an ideal high-performance packet processing model should be one that can be optimally mapped to performance-bounded targets. This resembles PISA pipelines – a sequence of fungible processing units with tight bounds on processing latency (to guarantee line-rate processing). The question is then about what an ideal (fungible) processing unit should be capable of, such that performance lies within the guaranteed bounds. More importantly, is there a way to programmatically synthesize such a unit (in both software and hardware) based on a specification. A language that maps intent to a performance-guided architecture is then the solution to the problem.

1.1.2 Vector Packet Processing

High-throughput packet processing is limited by memory access times, which even today is slower than network modulation rates at 100G. Hence, the only way to process packets at line-rate is through batching – a batch of several packets are processed all at once and then written to the respective buffers. Vector packet processing [21] (VPP) generalizes this concept with respect to packet processing frameworks. The most prominently known among them is the Intel data plane development kit [21] (DPDK). In this model, a user-space networking approach is used to bypass the kernel overheads of Linux networking stacks. Special NICs that allow user-space access to the NIC buffers are used to implement this approach. A user space process (usually pinned to a CPU core) constantly polls the NIC buffers to check for new packets, and subsequently copy a batch of packets to the (lockless) DPDK ring buffers. Leveraging cache locality, packets are then processed from there in fixed size batches before being enqueued for transmission. Complex operations on the packets, which require additional (and non-cache-aligned) memory accesses may slow down processing – a caveat that developers need to be aware of. The receiver and transmitter processes for the port can be scaled with CPU cores, allowing for high throughput packet processing. VPP enables line-rate packet processing on many general purpose servers equipped with traditional CPUs and memory. However, factors such as cache size and packet processing operations still constrain the maximum throughput that can be achieved. There are P4 targets that map to VPP frameworks such as DPDK, which aids developers in writing efficient packet processing programs.

1.2 Network Function Virtualization

General purpose servers provide a cost-effective, reusable and manageable infrastructure for running network functions alongside server applications building the case for network function virtualization (NFV). NFV takes an approach to running packet processing functionality on x86 (or equivalent) virtual machines. Packets are routed to servers hosting the virtual network functions and injected back into the network post processing. Load balancers, firewalls, proxies are some of the network functions that are implemented using this approach. NFV makes network functions manageable and scalable in the same way as regular server applications, making them ideal to deploy over general-purpose server infrastructure. Several frameworks and techniques to scale such an ecosystem have been proposed in the past.

E2 [69] is a scalable framework for virtual network functions with application-agnostic scheduling for network functions. It performs filtering based on ports and packet headers. Function placement is performed using policy graphs. Applications implemented using such a framework include network address translation (NAT), firewall, intrusion detection systems (IDS) and virtual private networks (VPN). NetVM [41] uses DPDK and KVM to partition memory regions for packet processing. It allocates each processing core with its own queue and network function, achieving 10 Gbps line-rate processing. It uses an address-based scheduling of packets to VMs. Function chaining is performed using shared memory region, trusted groups and dedicated chain forwarding cores. Applications implemented using NetVM include L3 routing, the Click [66] modular router and firewalls. OpenNetVM [94] uses DPDK and Docker to run network functions. It presents a scalable and modular approach to function chaining. It uses flow tables and TX threads to steer packets through chains and service identifiers to aid in the management of NF instances. Applications implemented include an IDS. OpenBox [13] presents a framework for development, deployment and management of network functions across VMs. It presents approaches to merging NFs and packet processing abstractions. It uses so called OBIs to implement NFs using metadata and session storage for maintaining state. It is able to achieve O(10us) latency and O(100Mbps) throughput. Applications include firewall, intrusion prevention system (IPS), web cache and a load balancer. NetBricks [70] presents UDF-based composition of NFs with packet processing abstractions. It performs chaining through function calls and run-to-completion scheduling. It uses zero-copy soft isolation using LLVM and unique types. Applications implemented using this framework include firewall, NAT, signature matching, monitoring and load balancer. LemonNFV [58] presents an approach to consolidation of NFs using PKU-based hardware isolation. It presents an approach to namespace-based isolation of NFs and so called trampolines for scheduling NFs. It also used software based fault isolation. Applications include IDS, NAT, ACL, connection tracking and DPI. OpenState [10] uses the concept of mealy machines. It uses OpenFlow messages to implement stateful processing on switches. Their API consists of XFSM tables to implement state machines and scopes based on ordered sequence of header fields. Applications include port knocking and MAC learning. OpenNF [30] presents a controller-driven approach to state management across NF instances. It uses flow-based state management with consistency guarantees. Examples include IDS, asset monitor, caching proxy and IPTables.

Various user-space and kernel-space optimizations have been created to cope up with increasing traffic demands within the network. Two major tasks include function placement and function chaining. Functions are placed across NFV hosts by considering traffic demands and network performance according to some optimal plan. Function chains are typically composed according to an operator-specified precedence. A large amount of work on NFV deals with performance optimizations including core-pinning, shared memory regions and hardware isolation. State synchronization across instances is another problem addressed by existing approaches, which gets more challenging as network speeds increase.

1.3 Programmable Packet Processors

Commercial network packet processing devices which allow programmable behavior have been around for a while, although they allow limited functional behavior to be configured onto the devices. However, recent advances in switching technology [1, 12, 42, 91] has enabled richer functionality to be programmed onto switching ASICs. These devices fall into one of two types of packet processing architectures – fixed-depth pipelined [12] and run-to-completion [42, 91]. The former has limited compute capabilities while their compute primitives typically guarantee line-rate forwarding performance. The latter allows more flexibility while allowing variable performance (e.g. below line-rate).

Run-to-completion processors such as Trio [91] can result in sub-line-rate performance. While there is enough redundancy in hardware to compensate for this, programmers must be aware of this caveat. Tofino switches guarantee line-rate performance once a program is compiled, but requires programmers to be aware of the resource constraints that guide compilation.

PISA switches have a fixed-depth pipeline of RMT [12] match-action processing stages, which are typically programmed using a P4 program. The compiler ensures that programs only compile if the resultant configuration can fit within the pipeline's resource constraints. Hardware restrictions determine what types of processing can be performed in match-action stages and how memory can be accessed – memory is local to each stage. However, stages are functionally redundant, i.e. any stage can perform any type of processing. This makes it a lot easier to compile programs using techniques such as ILP [47]. Such functional redundancy forms the bare bones for enabling active networking, as presented later in our approach.

To facilitate programming of such hardware, most vendors have supplied their own distinct interfaces making it difficult for users to interoperate across different types of devices. Building upon the fact that all packet processing tasks involve a common set of steps, P4 [11] attempts to overcome this hurdle by unifying device programmability with a language system. PISA devices, commercially available as Tofino [1] was the first to adopt this language as a means to program these devices. Since then, more device targets have been made programmable using P4 [42]. However, P4 is still target dependent – each device is required to export an architecture which is exposed through the language semantics. This makes P4 programs non-interoperable across devices, even though the language abstracts out most of the complexity of programming such devices. This makes the language appropriate as a low-level language [99] intended for systems programmers rather than application developers. Unlike languages meant for general-purpose CPUs, P4 programs for real devices need to take into account resource constraints on such devices, which are extremely limited; As compared to a commercial server machine which has several gigabytes of memory, a Tofino has a few tens of megabytes (of SRAM). Moreover, for PISA devices that follow an all-or-nothing approach to compilation, the task of compilation is both computationally hard [47] and slow. From a programmer's perspective this is indeed frustrating, since programs that are both syntactically and semantically correct may not compile due to resource constraints, significantly slowing down development time.

There are several concerns that govern the practical utility of such an ecosystem, one of which is multi-tenancy, i.e. multiple functions (e.g. forwarding, mangling) need to be run on the same device. Recent advances in the language [68] allow for a certain degree of modularity in programming through the use of control blocks. This has been leveraged by recent approaches [53] to compose (annotated) programs to a certain extent. To make the most efficient use of resources (while taking into account modularity) special data structures were introduced [40] for P4. Such approaches do not however, allow the parser to be sliced among modules. Other approaches have however, attempted to solve this problem using tags [96] or leveraging bit-slicing to move parsing functionality to the match-action processing units [79]. All of the above approaches present themselves as modifications to the P4 language or templating (e.g. using annotations). There have been other language-based solutions to program PISA devices, particularly for stateful programs (e.g. Domino [76]).

While the flexibility of run-to-completion devices make them attractive for implementing a large range of functionality, the performance-guided constraints of pipelined devices such as Tofino eases the burden on programmers to restrict their focus on functionality rather than performance. Is there a way to combine the best of both worlds – i.e. facilitate performance-aware programmability – for such devices? Moreover, can the ecosystem of programmable switches be modified to enable multi-tenancy?

The approaches presented here attempt to solve the above problems using prototypes built on top of PISA devices. The arguments presented here can serve as the basis for future improvement of the ecosystem of programmable switches and in-network computation in general.

1.3.1 PISA

A programmable switch ASIC like the Tofino [1] is a well-known target for the P4 programming language and is capable of processing packets at the rate of terabits per second. Such ASICs implement the protocol independent switch architecture (PISA) through reconfigurable match tables (RMT) hardware [12]. Packets arriving at physical ports of the switch are fed into heavily optimized pipelines of hardware units which process (and modify) the packets before sending them out on the wire. An ingress pipeline processes packets directly arriving on the physical ports while an egress pipeline processes packets as they move through egress port queues.

A traffic manager sits between the ingress and egress pipelines and is responsible for switching packets. A deparser concludes packet processing on the pipeline and reconstructs packet headers before sending it to the traffic manager or out on the wire. Tofino allows packets to be re-circulated back into the switch processing pipeline for further processing. Although powerful for implementing complex functions, it does come at a cost to bandwidth.

RMT. Reconfigurable match tables [12] (RMT) is an architecture for implementing match tables over programmable network processors. These match tables are implemented over a set of SRAM blocks that can be partitioned into logical tables. A set of hash units connect data from packet buffers into these SRAM blocks to perform key matches on data stored in the SRAM. Typically exact matches are performed over SRAM data. Ternary matches are implemented using TCAMs that can be reconfigured to logical tables in a similar way. The match results are fed to a set of action units which perform programmed operations on packet fields.

Parsing. Packet parsing follows a state-machine model: information extracted in a parsing state is used to determine the next parsing state. Header information extracted during the parsing phase is stored inside an internal data structure known as the packet header vector (PHV). The PHV can also be used to store additional information—in the form of metadata—during program execution. The PHV persists throughout the lifetime of the packet inside the switch.

Match-action processing. A series of identical match-action stages is responsible for implementing the core functionality of a P4 program. The match units in a match-action stage are stored in on-chip SRAM or TCAM and capable of performing exact or ternary matches. A set of primitives defined by the architecture can be invoked through abstractions known as actions. These primitives include arithmetic, logic and assignment operations along with a set of fixed functions such as hashes, counters and random number generators. A set of crossbars connect the data paths between match and action units. Additionally, Tofino enables read-write access to on-chip SRAM from the data plane through primitives known as registers. These features are exported to the P4 language through architectural constructs (known as "externs" in P4-16). Tofino registers are associated with ALUs which can perform a range of operations on register values.

The architecture of PISA switches has implications on the efficiency of resource utilization. Tables that require more memory than is available in a single stage can be chained across stages with the assistance of the compiler. However, there are other restrictions that may limit the size of tables altogether (e.g. the use of TCAMs). For programs that have dependencies (e.g. match dependencies), resources such as memory may not be available until later on in the pipeline. For a fixed pipeline architecture this leads to a wastage of resources. Alternative designs have been proposed such as dRMT [15] that use memory pooling to address such limitations.

Even then, resources that are once tied to a program cannot be altered without causing network disruption – another key limitation of the current ecosystem of programmable switches is the inability to perform hitless updates and modular programming. While several architectural designs have been proposed [25, 83, 87] that provide hardware extensions to support such capabilites, none have yet made it to commercially available switches.

1.3.2 Resource Sharing

Although P4 enables programmers to express a variety of functionality using a uniform set of semantics, it has limited support for multi-tenancy. Programmers have to manually write complex programs that contain functionality corresponding to multiple programs. Prior efforts have focused on hypervisor-based approaches [36, 93] to running multiple applications simultaneously on programmable switches: Hyper4 [36] allows runtime re-configuration of functions and allocated resources using control-plane support. However, these approaches do not support stateful processing. Moreover, supporting multiple stateful functions on a single P4 target necessitates sharing the limited resources (particularly memory) available on these devices. While devices such as Tofino have constructs (e.g. register ALUs) that allow isolation of memory regions, they have hard limits to the number of instances that can be created per stage and also cannot be allocated dynamically.

Since P4 targets can only run one program at a time, there have also been several efforts [39, 79, 96] to compose P4 programs in a modularized way. Our approach obviates the need to compose programs by decoupling programmability from resource allocation. Stateful programming only requires mapping of switch SRAM to programs, which we demonstrate can be done effectively at runtime.

1.3.3 Architectural Extensions

Even if a suitable binary were readily available, reprogramming currently available switches disrupts network processing. Researchers have proposed alternative architectures [83, 87] to allow incremental updates to the device without disrupting packet processing for the entire switch. 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 [83]. Unfortunately, such extensions have not made their way into commercially available devices.

Other proposed extensions seek to improve resource efficiency. For example, dRMT [15] 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 [87] 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) [25] 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 re-configurations. 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.

1.3.4 Virtualization

Our work is similar in spirit to prior attempts to virtualize standard RMT devices [36, 93, 98]. Like ActiveRMT, Hyper4 [36] 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 [36, 93].

Virtualizing a subset of switch resources such as stateful memory is a more tractable approach on current devices. NetVRM [98] virtualizes register memory constructs on programmable switches such as the Tofino. Memory is dynamically apportioned across a pre-compiled 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 fixed 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.

1.3.5 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 [98] attempts to determine the appropriate allocation using knowledge of utility gradients and network traffic. Determining an appropriate utility function is not always straightforward, however [31]. Take the example of a telemetry service such as the count-min sketch [17]; 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 determined 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.

1.4 Language

Programming emerging packet processors can become a tedious task in the absence of proper abstractions. P4 [11] addresses this problem for a large set of programmable devices. It is the most widely known language that can be used to program various targets including PISA and other emerging programmable switches [42]. The programming model consists of a series of constructs – parsers, control units and deparsers – that mimic a packet processing pipeline. These abstractions compile to device-specific resources that can implement the corresponding logic. Programmers can define custom header formats and logic describing how to parse them. Using information extracted from the headers, various functions can be implemented that make use of the processing units on the device. While most of the abstractions in P4 are target-independent, the ecosystem of devices are quite heterogeneous requiring special language modules to enable several capabilities. For this reason, P4 requires distinct architectures for each class of device targets (e.g. Tofino, BMV2, etc.). Some examples of target-specific capabilities include register memory and hashing – these are defined within their respective P4 architectures. P4 toolchains are hence associated with a target-specific compiler that implement the “backend".

The P4 language is however, limited in its support for modularization. For example, only one parser can be defined for a program. Control blocks do however, allow for some degree of modularization. A control block takes as input a set of headers and various metadata fields. A set of tables and their invocation order is defined within such control blocks. Additional imperative-style logic can also be implemented within the control blocks. Control blocks can be invoked recursively, making large programs more manageable. However, control blocks only simplify programmability – the actual feasibility of the program is determined by the compiler upon mapping the logic to resources. The simplest form of modular composition from multiple sources can be achieved using control blocks. Each programmer simply defines a control block based on a fixed set of headers and metadata.

P4 initially did not have much support for stateful processing. Language alternatives have been proposed in the past that attempt to address some problems such as stateful processing. Domino [76] is one such example. An imperative-style language is used to define operations on stateful memory on programmable devices. The idea is based on atomicity of operations. Every operation on stateful memory as defined in the corresponding packet transaction is atomic. This helps address the problem of data consistency. The complexity of such operations are however, limited by the number of required processing cycles to keep up with line-rate processing on a number of target devices. The Domino model is present on current RMT switches such as Tofino in the form of “register ALU" constructs.

1.4.1 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 [79] supports modular program composition by presenting a homogenized logical architecture and uses match-action tables to provide generic packet-processing capabilities. P4All [39] 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 [96].

While these approaches enable modular composition, they do not solve the chief drawback of P4-based approaches: the time to deploy new services remains dominated by compilation time. Moreover, RMT resource constraints (along with an all-or-nothing approach to ensure line-rate processing) frustrate compilers' attempts to map programs onto such devices, often resulting in compilation times on the order of minutes requiring sophisticated tools such as ILP solvers [47]. At times, such approaches even fail to find a solution due to their search strategy [76]. Chipmunk [28] 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.

1.4.2 Application Deployment Frameworks

Software-defined solutions to leverage all such hardware for network functionality has also grown over time. NFV frameworks [58, 69, 70] facilitate arbitrary network functions to be deployed over general purpose servers. They are however not directly applicable to programmable hardware accelerated data planes. This ecosystem is dominated by P4 [11] and its proposed enhancements [39, 76, 79]. While P4 does heavily simplify the task of expressing packet processing functionality, it is not sufficient in terms of multi-programmability – a necessary feature when using such an ecosystem to program a deployable network. Since the number of concurrent programs (functions) targeted for a single device are expected to be small (a few 10s), combining functions using static analysis seems to be a reasonable approach for most scenarios. Recent approaches have in fact been proposed [53, 96, 98, 99], that use such techniques to facilitate deploying functionality over a network of programmable devices.

ExoPlane [53] is one operating system that enables multi-tenancy for network functions deployed over a rack using resource augmentation. It attempts to provide an infinite switch resource abstraction by augmenting rack servers – equipped with smart NICs – with programmable switches. Applications defined in annotated P4 control blocks are composed into monolithic P4 programs and provisioned onto target devices. Packets are processed on a single device allocated by a resource planner which takes into account operator specified objectives. Packets are filtered on the programmable switch to determine whether they are to be executed on the switch or a rack server. Application state is synchronized across devices following certain (time-bounded) consistency guarantees. Such a system facilitates reactions to dynamic traffic patterns and increasing memory requirements.

Sirius [27] is a similar, yet more recent approach that composes P4 functions into monolithic chains using static analysis and spreads functionality across heterogeneous devices. It does so by leveraging redundant functionality across programs and defining gateways for each functional block. It also takes into account resource constraints to determine an ideal split of functionality across devices.

Both these approaches statically compose (templated) programs and target devices such as Tofino, where programs cannot be updated hitlessly. The limitations of efficient resource utilization, degree of multi-programmability and runtime programmability still remain with such an ecosystem unless alternatives such as virtualization is adopted. Moreover, some of them generalizes packet recirculations to run complex functionality – a feature that must take into consideration network bandwidth.

1.5 Active Networking

Our initial approach to service deployment harkens back to design patterns from classical active networking [9, 84, 86]. While the capabilities of currently available programmable switches allow [23, 85] 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) [44] 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 dissertation focuses on a subset of those issues, such as memory allocation and we defer the others to future work.

Other researchers have also considered employing capsule-based active networking techniques in modern switches. In one instance [44], the authors present an approach to expressing a range of telemetry functions by composing tiny programs (up to 20 bytes in length) using a small set of instructions. More recently, in-network telemetry [52] has been adopted as a standard for P4-based applications. Programmers can encode instructions in network packets according to a specific format in order to obtain a set of useful information from programmable switches. We provide a more expressive framework enabling a larger variety of applications.

Chapter 2

ActiveRMT

Numerous applications have been built around programmable switch hardware in recent times, especially with the emergence of reconfigurable match-action tables (RMT). This new class of devices is capable of performing basic computation and line-rate forwarding at a reasonable cost. Given this transformation in commodity switching functionality, we suggest it may be time to reconsider the concept of active networking, where end hosts can off-load application functionality to the network in real time without requiring the assistance of the network operator. We present a preliminary approach to encoding (nearly) arbitrary computation into a series of network packets that can be decoded and executed on programmable switch hardware. Our programs can leverage both high-speed forwarding and stateful capabilities of RMT devices. We also conduct an initial exploration into the importance of dynamically allocating switch resources across active programs to improve aggregate performance.

Researchers have proposed a wide variety of ways to leverage RMT-based programmable switching to offload functionality from end hosts, often taking particular advantage of the unique topological advantages afforded by in-switch processing. Recent systems demonstrate performance improvements in domains as varied as data aggregation [57], machine learning [74], object caching [46], distributed consensus [18] and network telemetry [33, 52] among others. The sheer number of disparate target domains and velocity of evolution of the associated P4 [11] ecosystem suggest there remain many additional benefits that are yet untapped.

We observe, that the capabilities of the current, P4-based deployment model fall considerably short of earlier visions for in-network computation. In particular, P4 application logic is statically deployed by operators at network configuration time. Despite the availability of programmable switch hardware from a variety of vendors [1, 91], there is considerable uncertainty regarding the viability of operator-managed service deployment. The P4 [11] ecosystem has enabled the development of a vast spectrum [38] of in-network services that can be compiled and installed on programmable switches such as those based on the reconfigurable match table (RMT) model [12]. 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 [6], 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 [14, 37].

Active networking [9, 84], in contrast, sought to enable user traffic to execute arbitrary programs inside the network, treating the switching fabric as a general-purpose execution environment. This more flexible approach provides the potential to dramatically expand the set of applications and users that can benefit from switch-based processing present in the network. Unfortunately, commercial switching hardware of the late 1990s and early 2000s was unable to provide line-rate forwarding performance while executing active programs, presenting a significant barrier to the adoption of proposed active-networking technologies. In subsequent years network processors were developed that are able to deliver line-rate network performance while allowing rich functionality, but their cost remains prohibitively high when compared to commodity switching gear. The increasing commoditization of RMT-based hardware, however, suggests price may no longer be a significant barrier.

We argue that the time is ripe for a reconsideration of active networking, in particular to study whether RMT hardware may finally unlock the promises of the original, traffic-directed computation vision. As a starting point, we show it is possible to use P4 to turn a commodity RMT switch (a Barefoot Tofino Wedge100BF-65X) into a something akin to a virtual machine, where program instructions contained within incoming network packets are executed while the packets are forwarded through the switch, potentially impacting not only the contents and forwarding behavior of the packet itself, but also the state of the switch—which, transitively, may impact the forwarding behavior experienced by subsequent packets.

Each packet can therefore perform a piece of computation that contributes to the overall functionality of a corresponding off-loaded application. We leverage the storage capabilities of programmable RMT hardware to maintain application state across packets and deliver complex functionality impossible to describe within a single packet.

Our approach naturally enables multi-processing within the switch. Because each incoming packet can contain its own, independent program, a switch may execute instructions for multiple programs concurrently. Importantly, unlike the current P4 model, where the network operator must determine the set of applications to support in a given switch *a priori*, in our model a new application can be executed on a switch simply by sending traffic containing that program—the switch does not need to be reconfigured in any way. Obviously, our model raises all of the standard questions regarding resource isolation and scheduling, namely defining both mechanisms and policies for protection and sharing. We defer almost all of these to future work, instead choosing to focus on the pragmatic question of feasibility: specifically, is it possible to replicate the functionality of recently-proposed P4 programs in our model? And, critically, can multiple such programs execute concurrently on a single switch while retaining their inherent performance benefits? For the particular applications we start with, the latter question comes down to one of resource management—can we dynamically allocate on-switch memory resources among competing programs to achieve better overall application performance?

We describe the space of programs that can be expressed with our programming model

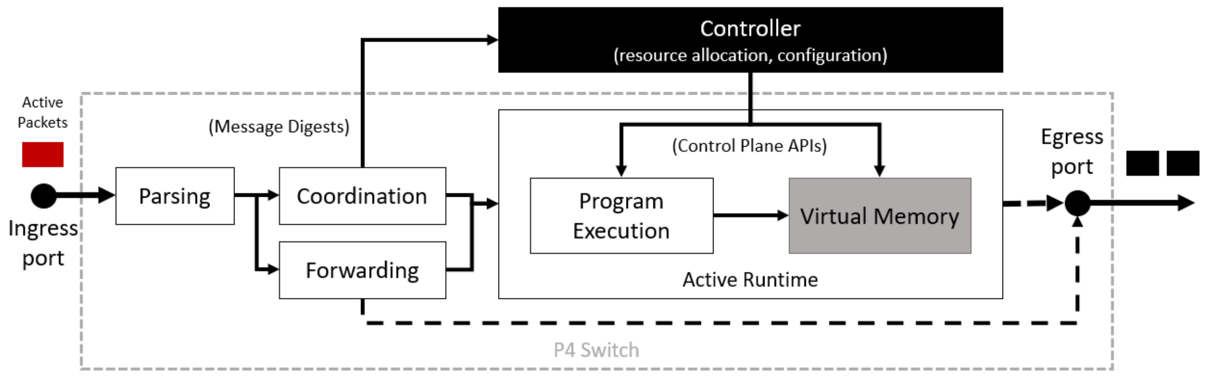


Figure 2.1. ActiveRMT packet processing overview

and introduce two example programs selected from the set of applications that has been implemented using P4. We choose one commonly used network function—a load balancer—as well as an example of application offload: in-network caching. Finally, we demonstrate the importance of the memory manager that resides in the control plane of the switch. In particular, we illustrate the impact of initial mechanisms that guide and enforce memory management on the performance the network applications are able to achieve. We conclude with a discussion of the remaining technical challenges that must be addressed before active networking might finally become a reality.

2.1 Overview

The RMT [12] 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 [11] and Domino [76] can be used to write programs that map to such configurations to enable desired behavior. We overlay a homogenized logical architecture (shown in Figure 2.1) 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. (Section 2.4 contains the details of our instruction set.) Programs can be attached to individual packets to trigger desired behavior when the packets traverse the switch.

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 [98] we are able to dynamically partition memory—both vertically (within stages) and horizontally (across stages)—resulting in more efficient resource utilization.

2.1.1 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) [5] and provides capabilities such as hashing and access to ALUs.

Instruction interpretation. Figure 2.6 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 2.1.5), instruction opcode, contents of the variables, and additional control flags. Table entries define valid memory regions for each program and are

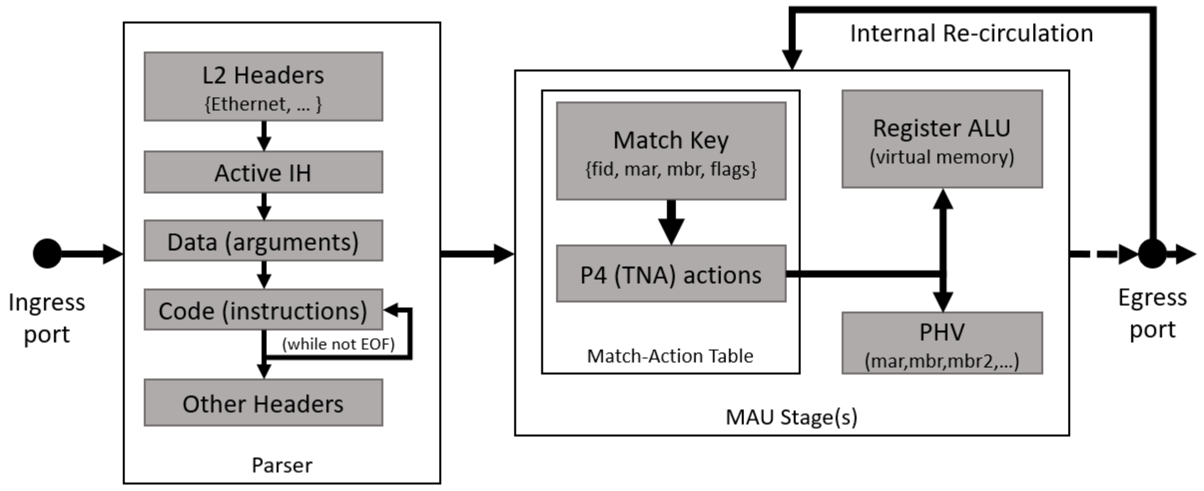


Figure 2.2. ActiveRMT runtime processing model

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.

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.

2.1.2 Memory Semantics

ActiveRMT uses a novel approach to achieve memory protection using TCAMs. Each instruction processor table performs a ternary match on the `MAR` register to determine if a memory access instruction can be executed, and only executes the instruction if a corresponding entry exists for the given *FID* and `MAR` range. The ternary match entries are populated according to the memory allocation for the corresponding *FID*. *FID*s essentially behave as memory protection keys, assigning programs to domains. These keys are assigned to programs one-to-one at the time of resource allocation. An alternative presumed authentication mechanism ensures that keys exclusively belong to one user when issued. Over the course of time, these keys can be reused. However, while keys are pinned to a particular program (and corresponding user), the respective allocations can change over time to make most efficient use of switch resources.

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 Section 2.4.4.)

2.1.3 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 (Section 2.6). 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.

2.1.4 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

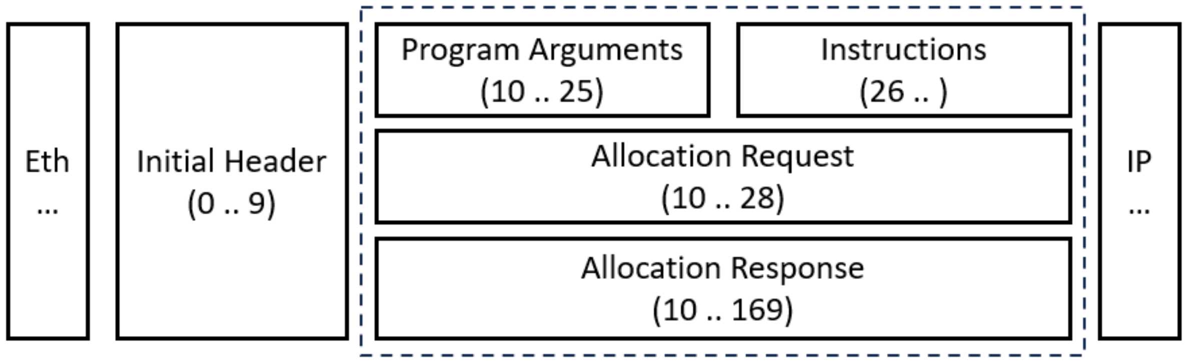


Figure 2.3. ActiveRMT packet header format. Numbers in parentheses indicate field byte position.

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 (Section 2.4.6) 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`.

2.1.5 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

[1]Such a design adds overhead. We could potentially optimize our solution by caching code on the switch; We present a similar approach later on.

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 POSIX networking stack. Figure 2.3 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 2.1.1). 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.

2.1.6 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 [90, 2, 89] on a PISA switch. (Additional examples of active

```
 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 2.1. Active program for querying an object cache

programs can be found in Section 2.6.) 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 2.1 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 3.1.3).

2.1.7 Object retrieval.

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

2.1.8 Data-plane cache management

Deploying a full-featured in-network cache service like NetCache [46] 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 most-frequent key-value pair among the set of keys that hash to each bucket [7, 67, 77]. Section 3.2.3 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 4.3.4 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.

2.2 Managing Switch resources

One of the key challenges raised by active networking is determining how to multiplex scarce switch resources across programs, not all of which may be executing at a given time (e.g., some programs may require continued use of stable switch storage even when none of the application's packets are transiting the switch). In particular, the performance of many of the applications that have been proposed for off-loading depends greatly on the resources available to the application at the switch. Here, we consider two of these resources: bandwidth and memory.

2.2.1 Limiting recirculation

While a 64-port switch with 100-G ports has enough backplane capacity to process 6.4 Tbps of traffic, packet recirculation changes things completely. A few recirculations can have significant effects on forwarding performance due to both available processing capacity and queue space. Larger and more complicated active programs are likely to require multiple passes through the switch and, hence, recirculation. However, as datacenter networks frequently have a surfeit of switching capacity it may make sense to transform this surplus bandwidth into useful computation. We attempt to manage this tradeoff in our prototype by implementing a recirculation manager for active programs. Our runtime monitors traffic levels on the switch using traffic meters based on the standard three-color marking scheme. Significant changes in traffic levels are reported to an application running on the switch control plane, which uses this information to dynamically assign limits to the number of allowed recirculations for active programs.

2.2.2 Memory allocation

Unlike bandwidth, which may be readily available in many environments, memory in the form of SRAM is likely to be scarce on most switches. While a single application using the switch can liberally use all the memory available on the device, careful partitioning of the memory space is required to enable multi-tenancy. Since the set of applications using the switch resources vary over time, memory management needs to be dynamic and automated based on predefined policies. In our prototype implementation of the memory manager, we explore a strawman approach of allocating memory equally to applications sharing the switch. Memory is explicitly requested by each new application, following which the memory manager reassigns memory equally among the new set of applications. For applications that benefit equally from memory, this approach enables fair sharing of resources.

To evaluate this hypothesis, we performed an experiment where we used four, separate caching applications (similar to the one we described in Section 2.3.4, but more full-featured) that share the switch over time. The keys corresponding to the requests are drawn from a Zipf distribution with $\alpha = 2$. All four applications first avoid the switch cache and directly access objects from the back-end key-value store. Then, each of these applications leverage the switch cache, one by one, over time. Figure 2.4 shows a timeline of the response latencies of application-level requests, averaged over one second. In the absence of on-switch caching, the destination server is able to respond to the requests in just over 6 microseconds. As we can see, the latencies for each application flow converge over time as the memory manager equally allocates memory among them—with latencies increasing slightly as each arrives.

Of course, the performance impact of decreased resource allocations varies across applications. Figure 2.5 shows the relative performance benefits of allocating varying number of stages to both the load balancing application (from Section 2.3.4, in blue in the figure) and the cache application (shown in orange). In this experiment, both applications are running concurrently (i.e., the incoming traffic consists of packets carrying object requests and flows to

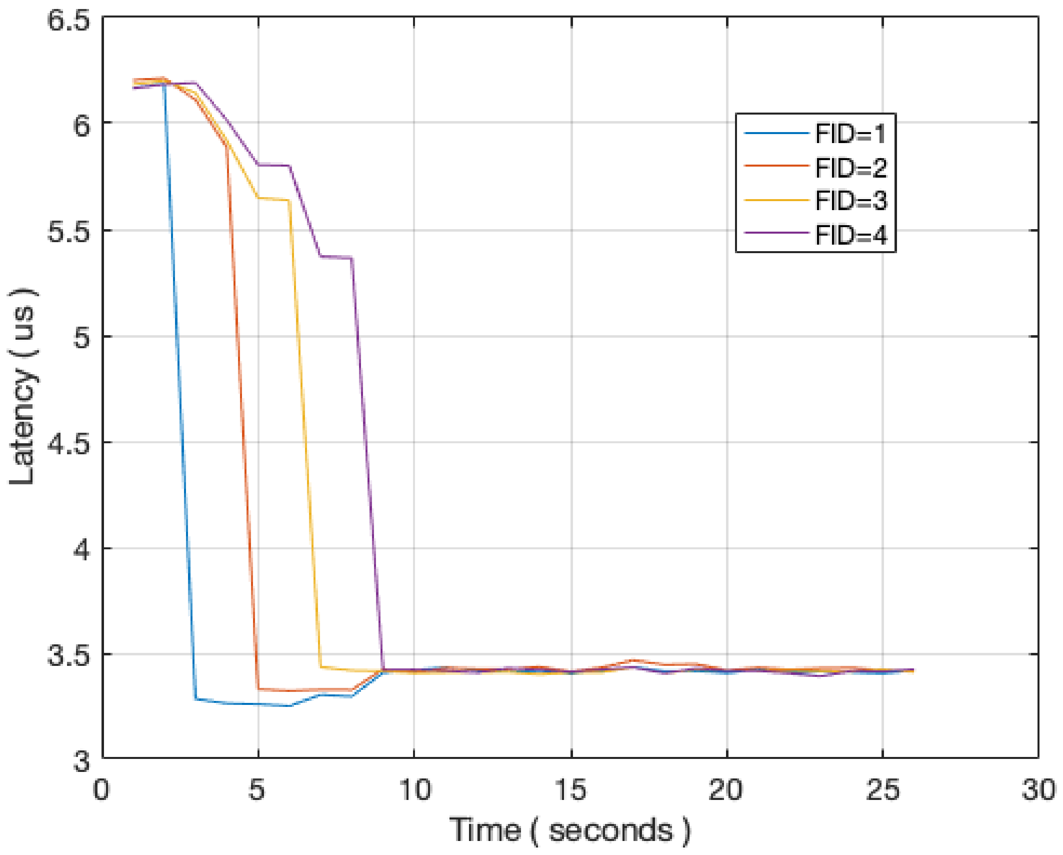


Figure 2.4. A demonstration of dynamic memory allocation. Initially, all four flows bypass the switch cache. Then, each flow begins using a separate active caching application that attempts to serve requests from on-switch memory. As more active applications arrive, per-application allocations decrease, resulting in a higher miss rate and increasing latency.

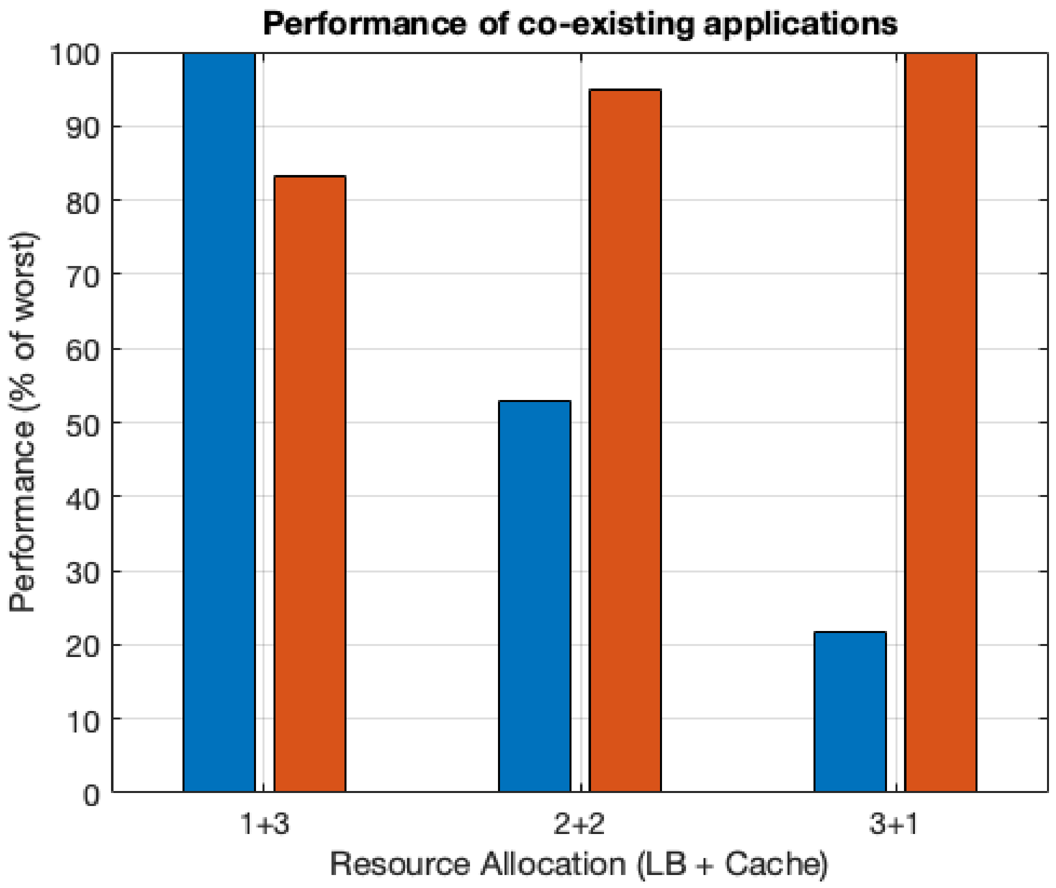


Figure 2.5. Performance benefits of memory allocation. We allocate four switch stages' worth of memory across two applications in varying proportion. The normalized load balancer drop rate is shown in blue, and response latency in orange.

be load balanced), and we consider three different allocations of four switch stages worth of memory. The left-hand grouping shows when one stage is allocated to the load balancer and the remaining three to the cache, the center evenly across applications, and the right-hand bars consider the reverse. The bars plot an application-specific performance metric (request latency in the case of the cache and flow drop rate in the case of the load balancer) normalized to the worst-case performance where each application is allocated only one stage. While the absolute heights of the bars are incomparable across applications, in both cases lower is better. One can deduce that the load balancer extracts greater relative performance benefit than the cache from a commensurate memory allocation.

2.3 Programming Model

Our active programming model is built on top of the semantics exported by the P4 programming language and the associated RMT architecture. An active program consists of a sequence of instructions contained within one or more packets. Execution follows the von Neumann model, where instructions are executed sequentially: once an instruction is executed by any switch in the network it will not be executed again. In our initial work, we consider programs that run to completion inside of a single switch (possibly taking multiple passes through the egress pipeline) but we imagine it may be preferable for packets to execute across switches in networks with multiple RMT-capable devices.

2.3.1 Instruction processing

Figure 2.6 shows the relationship between match-action tables, packet and ephemeral metadata contents (stored inside the PHV) and switch storage and computational units. Arithmetic, logical and bit-shift operations are enabled by the action units in the RMT pipeline. Instructions and their arguments are decoded by the match tables and executed by the corresponding action units. The match tables use the opcode (loaded by the parser) to determine which action to execute. Space prohibits us from listing a complete set of operations, but we illustrate several in the examples that follow.

The runtime uses a variety of packet metadata buffers to facilitate the control flow of the program. Data flows between packet header fields and these metadata buffers during program execution. The instruction set architecture exposes these buffers as a fixed set of general-purpose 'registers' that are available for use by operations in the instruction set. These registers can be loaded from and stored to both packet header fields and persistent memory on the switch through a set of memory access instructions. Two registers of particular interest are the memory address register (`MAR`) and memory buffer register (`MBR`), so named to reflect the semantics of the corresponding registers in a general-purpose CPU. The instruction argument is used to pass

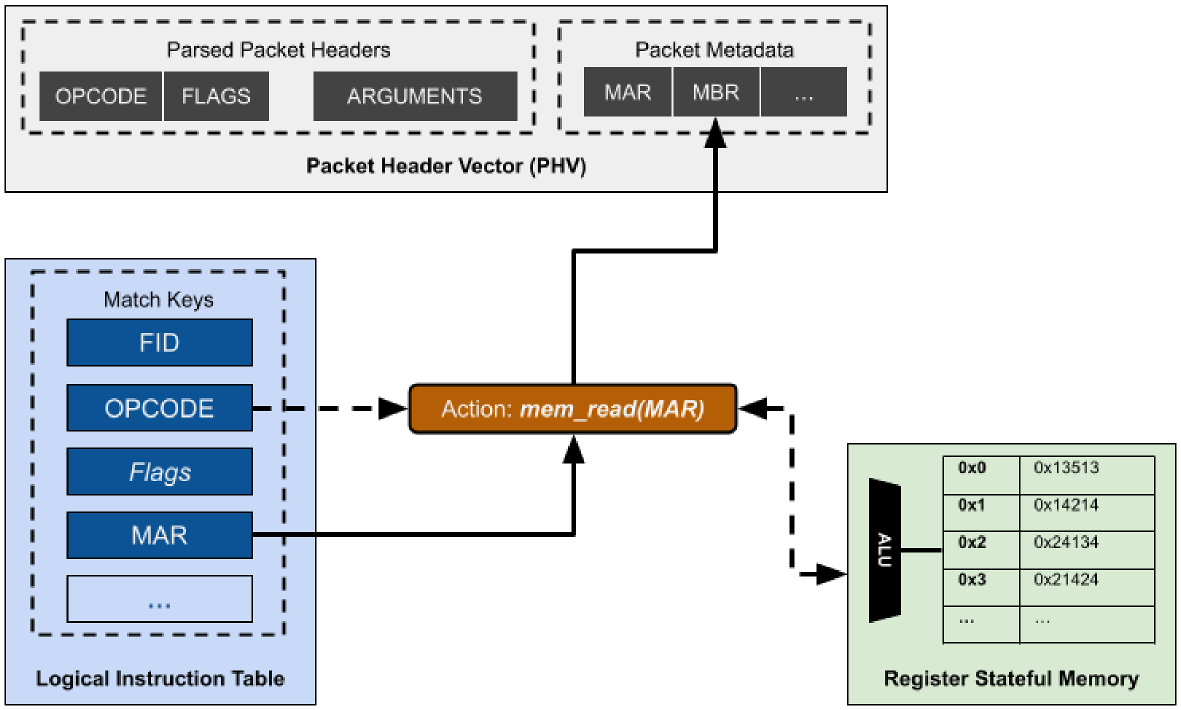


Figure 2.6. An example of executing a `MEM_READ` instruction; the vale stored at `MAR` in the switch's stateful memory is loaded into the `MBR` register.

information from the packet directly to the action unit. The `label` is also loaded by the parser and is used to mark locations in the program to which it may jump.

2.3.2 Memory model

Persistent switch memory is somewhat different than typical architectures and comes in two forms: directly addressable, and associative collision chains whose length is dictated by the number of match-action stages in the switch hardware. Direct memory locations are specified by 16-bit addresses, while values stored in collision-chain memory are accessed through 16-bit-wide keys.

In contrast to registers which are available throughout a program's execution, the persistent memory locations available to an instruction depend upon the hardware stage on which the instruction is executed: both memory addresses and keys are scoped by the current pipeline stage. Said another way, the location of an instruction in a program dictates which portions of memory it can access, as memory physically located at one match-action stage cannot be accessed from another stage. This RMT-based restriction currently necessitates some careful program construction, but could potentially be alleviated by clever compiler design: Memory in later stages can always be accessed by inserting the necessary number of `NOP` instructions before the access, while locations on previous stages require re-circulation back into the egress pipeline to execute the instruction at the appropriate stage in the next pass.

Memory protection is enforced by the runtime during program execution. Allocation policies defined in the control plane allocate regions for corresponding applications. Memory re-allocation is triggered whenever the set of tenant applications change. An incorrect memory access by an instructions results in the generation of a segmentation fault, where the corresponding active packet is returned to the sender with a specific bit set in the active program header. The application then retrieves the allocated memory region from the switch runtime.

2.3.3 Control flow

Our current instruction set supports both branching and a basic loop construct. Apart from loops, our runtime currently cannot re-execute instructions already executed previously. In other words, one can only jump forward in the program. This can of course be partially compensated for by repeating instructions as necessary, but a practical implementation would likely remove this restriction.

Branching

A jump instruction branches to label in a different part of the program while a return instruction terminates execution of the program and branches out of the program. `UJUMP` is unconditional, while `CJUMP` checks the value of `MBR` and jumps if the value is not zero, while `CJUMPI` jumps if the value is zero. Unlike typical execution environments, however, active programs have constant-time execution despite the presence of conditional branches. All branches of a conditional are visited irrespective of the condition. Instructions in inactive

```
1. MAR_LOAD, 0x0001
2. MEM_READ
3. CJUMP, goto=2
4. MBR_ADD, 1, label=2
5. MEM_WRITE
6. RETURN
```

Listing 2.2. Conditional example

branches are simply ignored (i.e., treated as NOP) while the active ones are executed. Return instructions are used to terminate execution of the program. This is a mandatory instruction for every program. A regular RETURN instruction marks the program execution as complete immediately and can be invoked anywhere in the program. The CRET instruction returns only if MBR is non-zero.

We illustrate both memory access and branching using a trivial example shown in Listing 2.2 that increments a value stored in memory location 0x0001 only if the value is currently zero, and stores the result in a different location. The first instruction sets the value of MAR to the address literal 0x0001. The MEM_READ instruction then loads the value at that memory location (in stage two of the pipeline) into MBR. The CJUMP instruction causes the next instruction (number 4, labeled 2) to be skipped if the value of MBR is non-zero. Else, it executes the MBR_ADD instruction which adds the numeric literal 1 to the contents of MBR. Finally, the MEM_WRITE instruction stores the value in MBR back to the address specified in MAR—but this time into the memory bank of stage five.

Loops

A loop follows the 'do-while' idiom common in many higher-level programming languages. We explain how a loop is written using the example shown in Listing 2.3. The loop conditional that determines whether the next iteration of the loop should be executed or not is based on the value in MBR. A non-zero value indicates that the loop should continue. In the

```
1. MBR_LOAD, 3
2. LOOP_INIT
3. DO, goto=2
4. MBR_SUBTRACT, 1
5. WHILE
6. NOP, label=2
7. RETURN
```

Listing 2.3. Loop example

example `MBR` is loaded with a value of 3 to cause the loop to execute three times. Once this value reaches 0 the loop will terminate. The `LOOP_INIT` is used to initialize the loop by indicating that the next instructions are iterable. The `DO` instruction implements the loop conditional and checks whether the value in `MBR` is zero or not. The `goto` label specifies where to branch to if the loop is done. The target location in the program is specified by a `label`. In the example if `MBR` is zero then it jumps to the instruction labeled 2 (i.e., instruction six). Else, the body of the loop is executed. In this case the following `MBR_SUBTRACT` instruction is executed which decrements the value in `MBR` by 1. The `WHILE` instruction is the final instruction in the loop construct and indicates the end of the scope of the loop. Control-flow instructions such as the branches described in the previous section can also break out of the body of a loop.

2.3.4 Examples

We have used our programming model to implement a variety of network functions. Here, we present two toy examples based upon published applications implemented on RMT hardware using P4 to demonstrate our ability to express similar functionality in our framework. Then, in the subsequent section, we consider performance concerns raised by our model.

Stateful load balancer

Stateful flow-level load balancers have been implemented in the P4 ecosystem before [4] and require forwarding packets based on state. We wrote an implementation of a trivial load balancer using our active programming language that selects a random egress port to use for the duration of a flow; the program itself need only be included in the first packet of the flow—or whenever the network chooses to redirect the flow. (A useful load-balancing application, of course, is likely to select only among a subset of the egress ports, and may need to adjust the packet destination addresses while doing so.)

Listing 2.4 shows the function written using our active language. The instruction `LOAD_5TUPLE` loads the packet's 5-tuple (i.e., source and destination IP address and port, along with the IP next protocol field) into a data structure. The subsequent instruction `HASH_GENERIC` applies a hash function to the contents of this structure and stores the result in `MAR`. The instruction `RANDOM_PORT` selects a random output port and stores it in `MBR`. The `SET_PORT` instruction at step 9 updates the switch's forwarding table to set the egress port for all packets associated with this packet's 5-tuple (i.e., the current flow) to the contents of `MBR` (i.e., the result of the hash). The sequence of associative `MEM_WRITE` instructions in steps 4–8 attempt to write the contents of `MBR` to the memory location associated with the key stored in the `MAR` in each of the four stages following the semantics of a collision chain. (In particular, any successful `MEM_WRITE` will prevent the execution of any subsequent ones.) The `C_ENABLE_EXEC` instruction enables the subsequent (`SET_PORT`) instruction only if the associative write was successful. Packets that are unable to store their selected output port (due to an overflowed collision chain) are dropped in this trivial example.

Object cache

We draw inspiration from systems like NetCache [46] to implement an in-network cache using our prototype language. A request for an object received at the switch is first looked up in its local storage. If it is found then the object value is returned to the requester, else the switch

```
1. LOAD_5TUPLE
2. HASH_GENERIC
3. RANDOM_PORT
4. CMEM_WRITE
5. CMEM_WRITE
6. CMEM_WRITE
7. CMEM_WRITE
8. C_ENABLE_EXEC
9. SET_PORT
10. RETURN
```

Listing 2.4. Program for toy stateful load balancer

forwards the packet to its intended destination as usual. The cache uses the stateful memory on the switch to implement a key-value cache. Values are accessed in a collision chain across several stages. Listing 2.5 shows a program for reading objects from the switch cache. (Storing objects requires a separate, complimentary program which we do not include here due to space constraints.)

The first instruction loads the 16-bit key (specified in the packet as a literal instruction argument) into `MAR`. The next four instructions attempt to read the value associated with `MAR` using the semantics of a collision chain. (Any successful `CMEM_READ` instruction disables the execution of all subsequent ones.) The `ENABLE_EXEC` instruction is used to resume execution of the program once it is paused due to a successful memory access. The `CJUMPI` instruction performs a conditional jump if a cache hit occurs and executes the `RTS` instruction. This instruction marks the packet for returning back to the sender. The subsequent `ACC_LOAD` instruction is executed thereafter and the value of the `MBR` (which contains the object value) is loaded into the packet header. Otherwise, a cache miss occurs and the packet is forwarded along to its destination—presumably a server which can service the request.

```
1.  MAR_LOAD, <key>
2.  CMEM_READ
3.  CMEM_READ
4.  CMEM_READ
5.  CMEM_READ
6.  ENABLE_EXEC
7.  CJUMPI, 2
8.  RTS
9.  ACC_LOAD, label=2
10. RETURN
```

Listing 2.5. Program for toy cache read

2.4 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.

2.4.1 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.

2.4.2 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`.

2.4.3 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.

2.4.4 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`.

2.4.5 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`).

2.4.6 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.

2.5 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.

2.5.1 Switch runtime

Our P4-based runtime consumes 100% of SRAM available for register memory in each stage (75% overall per 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, which is only slightly less than native P4 and substantially higher than NetVRM. Native P4 programs 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. Due to its use of virtual address translation and constraining the total addressable memory region per stage to be a power of two, NetVRM is only able to make less than half of the match-action stage resources available to application programs [98].

2.5.2 Client compiler

An active program such as the one described in Section 2.1.6 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.

2.5.3 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).

2.6 Active Programs

With a total of 20 virtual processors mapped to RMT stages, with each capable of executing over 50 instructions, the number of programs in theory is astronomical. We however, attempt to express a few useful and interesting applications using our programming model that could benefit from offloading onto a programmable switch. These are described below.

2.6.1 Heavy-Hitter Detection (Cache)

Listing 2.6 shows the active program for computing frequent items. For our cache application described in Section 2.1.6 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.

```
MBR_LOAD // load key 0
MBR2_LOAD // load key 1
COPY_HASHDATA_MBR
COPY_HASHDATA_MBR2
HASH
ADDR_MASK
ADDR_OFFSET
MEM_MINREADINC
COPY_MBR2_MBR
HASH
ADDR_MASK
ADDR_OFFSET
MEM_MINREADINC
COPY_MBR_MBR2
MAR_LOAD
MEM_READ // read hh threshold
MIN
MBR_EQUALS_MBR2
CRETI
MBR_LOAD // reload key 0
MEM_WRITE
NOP
NOP
COPY_MBR_MBR2
MBR2_LOAD
MEM_WRITE
COPY_MBR_MBR2
MEM_WRITE
RETURN
```

Listing 2.6. Active program for computing frequent items for a cache with 8-Byte keys and 4-Byte values.

2.6.2 Cheetah Load Balancer

For load-balancing we adapt the P4 based approach to the Cheetah load balancer [4] 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.

Server selection

Listing 2.7 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 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

```
 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 2.7. Active program for SYN packets in CheetahLB.

line 27.

Flow routing

Listing 2.8 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

```
 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 2.8. Active program for non-SYN packets in CheetahLB.

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.

2.7 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.

2.7.1 Memory READ

Listing 2.9 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).

```
MAR_LOAD,$ADDR
MEM_READ
MBR_STORE
RETURN
```

Listing 2.9. Remotely reading a memory location.

```
MAR_LOAD,$ADDR
MBR_LOAD,$DATA
MEM_WRITE
RETURN
```

Listing 2.10. Remotely writing a memory location.

2.7.2 Memory WRITE

A corresponding memory write active program can be found in Listing 2.10. 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.

2.8 Caveats

While our programming model allows expressing a large number of programs, ActiveRMT comes with several caveats which must be taken into consideration. We describe these in this section.

2.8.1 Memory Consistency

Our memory semantics is based on register ALUs on Tofino. These units are individually programmable and can be used to perform a number of (atomic) operations on register memory. In order to achieve generality, ActiveRMT pre-programs them to export a subset of (common) operations – such as read, write and increment – that allow expressing a range of functionality.

However, more complex operations that require atomicity have to dealt with specially. Let's consider the following pseudocode for a conditional update:

```
if (x % 2) == 0:
    x = x + 1
```

This pseudocode can be implemented in ActiveRMT as follows:

```
MBR2_LOAD,0x0001    // Load mask = 1.
MEM_READ            // Read the memory value (assuming MAR has address).
MBR_AND_MBR2        // Apply the mask to check for divisibility by 2.
CJUMP,:LOC          // If not divisible (i.e. x % 2 > 0),
RETURN,@LOC         // then return.
NOP                 // Fill with NOPs to force re-circulate.
...                 //
MEM_INCREMENT       // Increment the value at the same stage.
```

The above raises a concern for memory consistency. Let's say that there are n pipeline stages in the active runtime. After the first packet reads the memory object at line 2 (let's assume the value is even), another n-1 packets would have read the same value before the first one updates it. This would result in the value of 'x' to be updated to `x=x+n` instead of `x=x+1`. Since this read-modify-write update to 'x' is not atomically feasible on ActiveRMT, we would need to use locks. Fortunately, we can use semaphores using ActiveRMT. The following is a pseudocode for the same program with locks:

```
lock(s)
read(x)
y = x & 1
if y == 0:
    <<re-circulate>>
    increment(x)
    <<re-circulate>>
    unlock(s)
```

The procedure `lock(s)` can be implemented as follows:

```
MEM_INCREMENT   // Read current value as "s" and increment.
CJUMPI,:LOC     // Jump if value == 0.
CONTINUE,@LOC   // Continue otherwise.
```

The special instruction `CONTINUE` terminates execution of the packet and marks it for re-circulation. This causes the packet to spin within the pipeline until the value 's' is cleared.

This approach however, requires two re-circulations (one additional to release the lock). For workloads that are write-dominant, this would consume significant bandwidth. More importantly, since upon re-circulation packets are added to the (e)gress queue, there is a chance of congestion-related packet drops. Hence, the program may not run to completion resulting in a deadlock!

To mitigate deadlocks, a watchdog program can be used which monitors the value of 's'. Note that this value is incremented when a new packet arrives or the previous one spins around. Either way, it is an indicator of the number of packets processed (k) and hence the number of cycles elapsed can be bounded by `k * T`, where T is the number of cycles per pipeline. A timeout can then be chosen appropriately. The following is a representative program:

```
MBR2_LOAD,$THRESHOLD    // Threshold for timeout.
MEM_INCREMENT           // Read semaphore value.
MIN                     // Store min of threshold and value.
MBR_EQUALS_MBR2         // Check if this is equal to the threshold.
CJUMP,:LOC              // "
NOP                     // Force re-circulate.
```

```
...                      // "
CLR_MBR                  // Set MBR to 0.
MEM_WRITE,@LOC           // Clear the semaphore.
```

2.8.2 Effects of Congestion

One must note that packets can get dropped at queues depending on the level of congestion. There is no way to avoid this solely at the switch. This also applies to re-circulated packets. Hence, programs are not guaranteed to run to completion. Even if priority queues are used, the problem can only be alleviated. This is one of the fundamental limitations of RMT.

However, such a limitation may not necessarily violate consistency semantics. Consider the cache read program described previously. The key may be read in the first pass through the switch whereas the value may be read in a subsequent pass. If the packet gets dropped after re-circulation, the result would simply not be returned to the sender and there would also be a packet loss over the network – the sender could then re-transmit the packet. When writing to memory, one could take care of consistency be ensuring that writes are idempotent – this is used in our example implementation of populating the cache.

In any case, such a limitation is inherent in the design of networks – which is itself a best effort service. While packet drops due to congestion cannot be avoided completely, one might attempt to workaround them through congestion control mechanisms and traffic engineering. For example, one might consider tracking queues within a switch to decide where to re-circulate packets. Typically, on devices such as Tofino, re-circulated traffic is assigned a priority distinct from regular traffic. While this may indeed affect run-to-completion, additional scheduling mechanisms may be used to avoid packet drops internally. Chapter 4 discusses mechanisms to handle such scenarios.

2.9 Discussion

The large emerging space of application functions that are being built on top of the P4 ecosystem makes it worth reconsidering the way these functions are executed on a P4 switch. It makes more sense for network behavior to be directly controlled by the application without having to interact with network infrastructure. Furthermore, thinking on the lines of virtual network functions, being able to multiplex functions in a manageable way appears to be a necessity. Our approach attempts to deliver the above using ideas from active networking introduced decades ago on recent programmable switch hardware. However, we must take into account the resource constraints on such hardware and shed some light on how effectively managing them could affect application performance and functionality. We focus on one particular scarce resource – stateful memory – in our next chapter.

Acknowledgements

Chapter 2, is a partial reprint of the material as it appears in – *Rajdeep Das and Alex C. Snoeren. "Enabling Active Networking on RMT Hardware". In: Proceedings of the 19th ACM Workshop on Hot Topics in Networks. Virtual Event, USA, November 2020, pp. 175–181.* The dissertation author was the primary investigator and author of this paper.

Chapter 3

Memory Management

Most practical applications require stateful memory, a scarce resource on programmable switches. Typically, such memory is statically allocated at compile time along with packet processing logic. However, 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. Building on top of ActiveRMT, 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.

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 [6, 25, 83, 87], software virtualization to enable multi-programming [36, 93, 96], and dynamic resource allocation among a fixed set of services [39, 98]. 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.

Earlier, we presented an alternative approach to service deployment that enables existing RMT-based hardware to support multi-programmability and dynamic resource management: *ActiveRMT* (Chapter 2). 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 [84]) 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 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 [46], load balancing [4], and network telemetry [77]—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 3.2.1). Time to deployment for new services is comparable (Section 3.2.2) to published reconfiguration times for approaches using hardware extensions [83, 87], and resource overheads are much lower (Section 2.5.1) than in prior virtualization approaches [36, 93]. Accommodating new services does not disrupt network operation: only services—if any—whose allocations were adjusted to make room are affected (Section 3.2.3).

3.1 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.

3.1.1 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

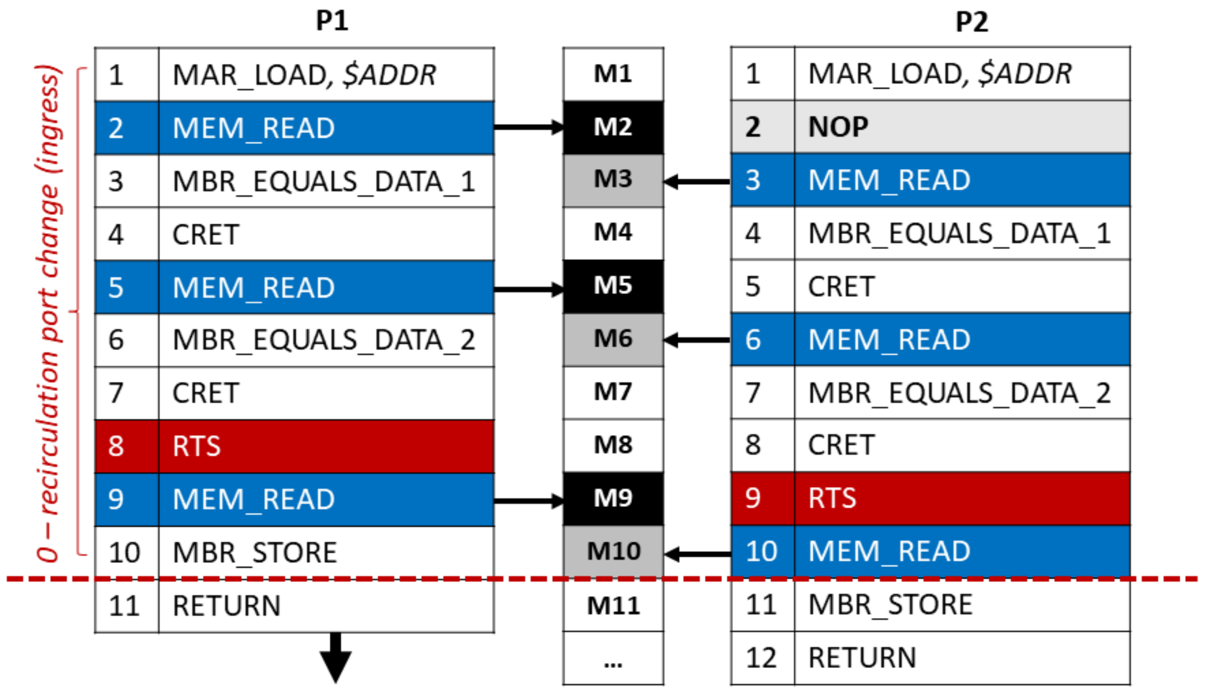


Figure 3.1. Mutating a program to efficiently fit within the available memory region. The red line marks the limit to which the RTS instruction can be moved.

each pool is tied to an execution stage, memory access instructions require an allocation in the corresponding stage. For the example program in Listing 2.1, 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 3.1 illustrates how we can mutate the cache program from Listing 2.1 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 3.2.1 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 [98], ActiveRMT allows memory regions of arbitrary size, but analysis (Section 3.2.4) 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 [98] 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 2.1.6 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.

3.1.2 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 [47, 39, 28]. 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 2.1 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 3.2.1 shows we can find solutions for our example applications rapidly in practice.

Allocation scheme.

We compute the cost **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 3.2.4.

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 [19, 50, 64, 71, 72] and attempt to deliver max-min fairness. Because memory is not arbitrarily divisible, we approximate it using progressive filling [72].

3.1.3 Allocation Process

Clients initiate an allocation with allocation-request packets (described in Section 2.1.5). 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 3.2.2 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.

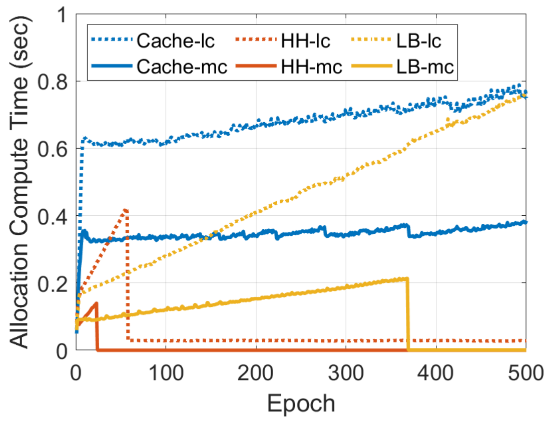
ActiveRMT provides two methods for a client to extract existing memory contents from the snapshot before its new allocation is applied and packets "reactivated." One is via the control plane (using APIs to access register memory) and the other is via the data plane (using active packets); we expect most 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 $\times$ 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. Section 2.7 shows examples of how to use these instructions; we employ them to populate the cache in Section 3.2.3.

3.2 Evaluation

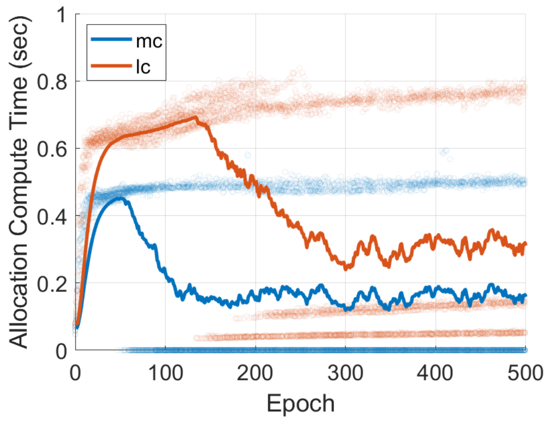
We evaluate the performance of our ActiveRMT prototype along two distinct dimensions: 1) the effectiveness of its online memory allocator, including the impact of (re)allocations on existing services, and 2) the speed with which we can provision new services. We also include a case study of a full-featured 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.

3.2.1 Memory Allocation

We evaluate the performance of our memory allocator when faced with different mixes of three active applications: an in-network cache [46] (as in Listing 2.1), stateless load balancer [4], and heavy-hitter detector [77] (with implementations in Section 2.6). 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.

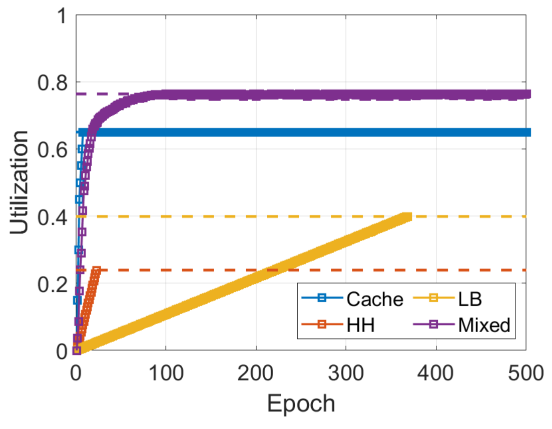


(a) Pure workloads

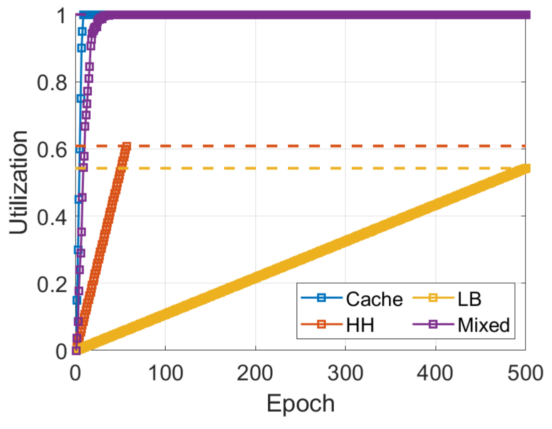


(b) Mixed

Figure 3.2. 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

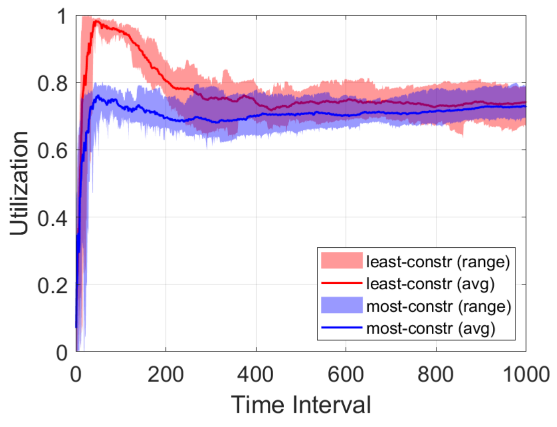
Figure 3.3. Memory utilization of four different workloads using (a) most- and (b) least-constrained allocation policies.

Computation time

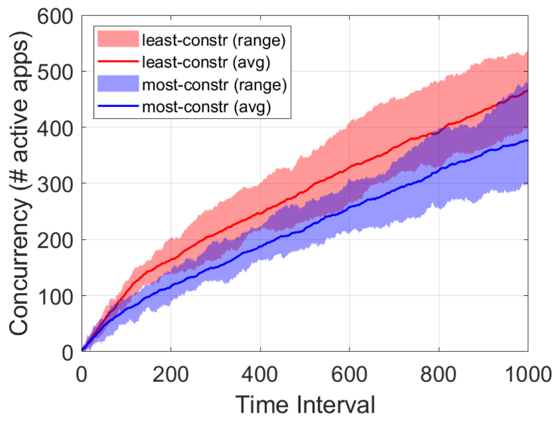
We start by ensuring that the computational task of calculating allocations is sufficiently modest so as to be performed by a switch's control processor. (For now we focus exclusively on control-plane operations and return to the data-plane aspects in subsequent experiments.) We consider two different 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 3.2a 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 ($O(10\text{ ms})$).

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.

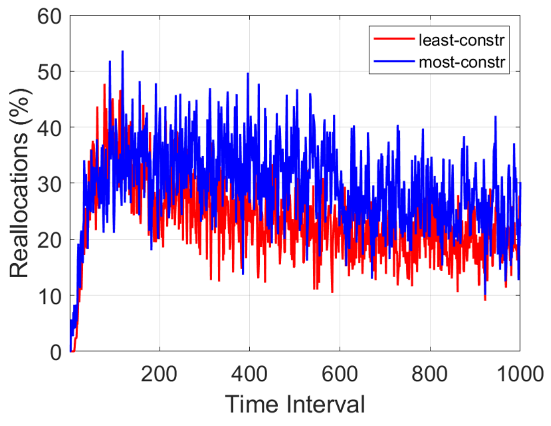
Pure workloads are unlikely in practice, however. Figure 3.2b shows the computation time for a mixed workload (where instances are chosen uniformly at random among cache, heavy-hitter and load-balancer applications). We plot the allocation time for each arrival for 10 random trials as a scatter plot. 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



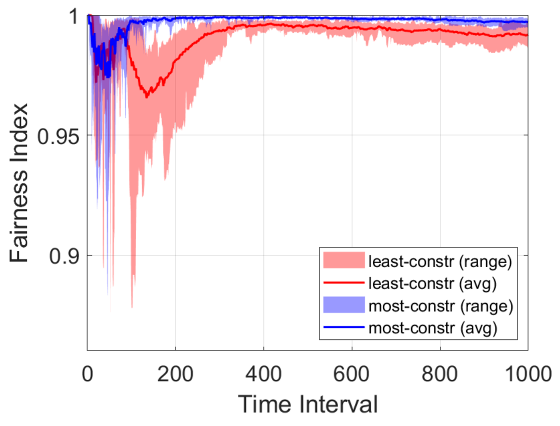
(a) Utilization



(b) Concurrency



(c) Reallocation %



(d) Fairness (elastic)

Figure 3.4. 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.

correspond to cache instances. In sum, while certain cases (e.g. purely elastic applications or a long run of exclusively load-balancer instances) 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 3.3 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 3.4a 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

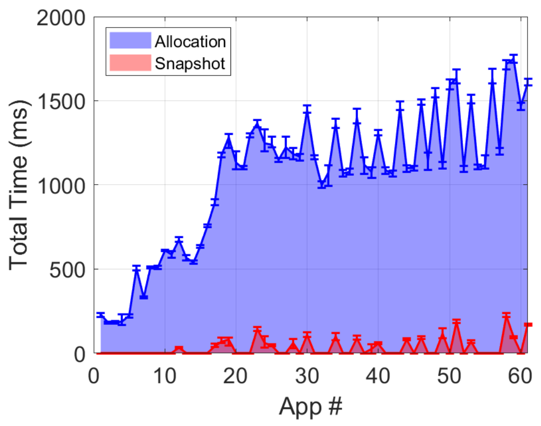
Figure 3.4b plots the number of resident applications in each epoch. As expected, the overall population grows over time, with the least-constrained policy able to place more instances due to its increased flexibility. Not all arrivals can be placed; after about 100 resident instances the allocator can 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

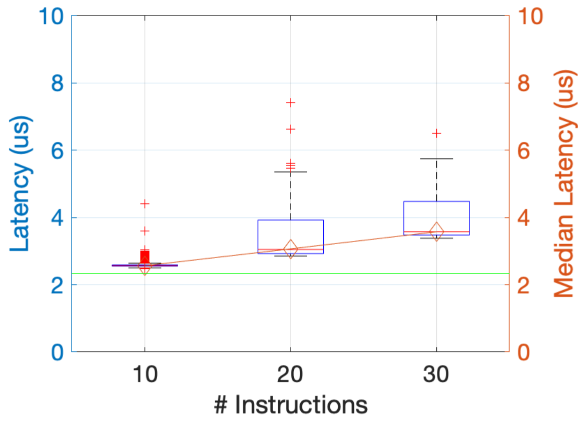
While inelastic applications are unperturbed by the arrival of new applications, elastic applications are likely to be reallocated with some frequency. Figure 3.4c shows the fraction of resident cache applications that are reallocated (alternatively, the expectation that any given instance will be reallocated) in each epoch in the same set of arrival sequences under both policies. To enhance visibility we plot the exponential moving average ($\alpha = 0.6$). The frequency increases initially but stabilizes after there are multiple cache mutants inhabiting each stage.

Fairness

Figure 3.4d plots Jain's fairness index [43] 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



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



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

Figure 3.5. ActiveRMT latency overheads on a Tofino.

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.

3.2.2 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 3.5a 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.

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.

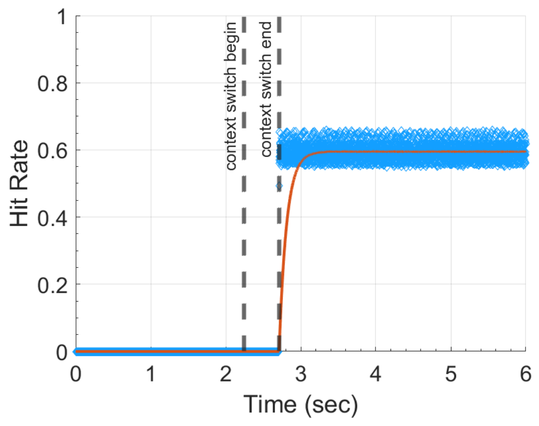
Processing latency

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 3.5b. 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.

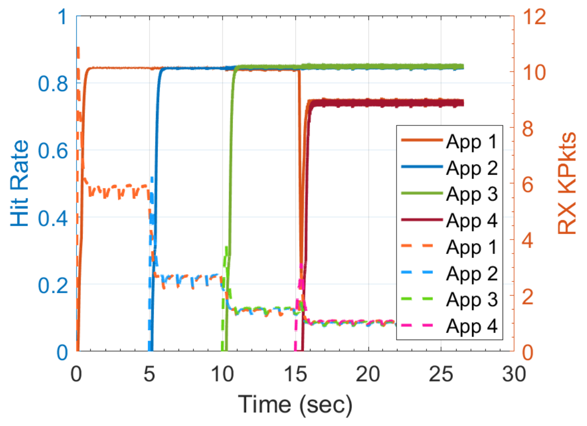
3.2.3 Case Study

Section 2.1.6 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 2.1 on its application-level requests.

Figure 3.6a 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, 89, 90] 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 3.1) and activates each of its object requests with this program (shown in Section 2.6.1). 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 [7] 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 3.1.3) 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



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



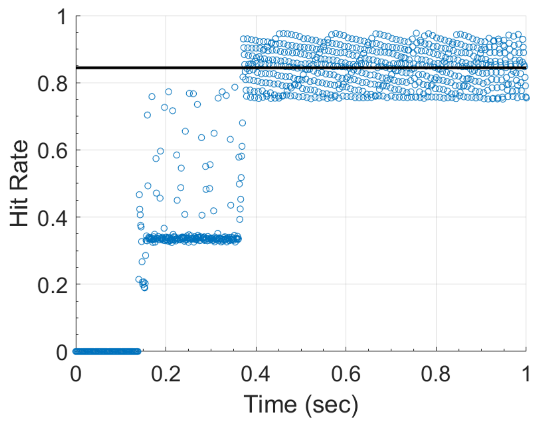
(b) Hit rate (solid, left axis) and throughput (dashed, right axis) of four cache instances.

Figure 3.6. Full in-network cache lifecycle.

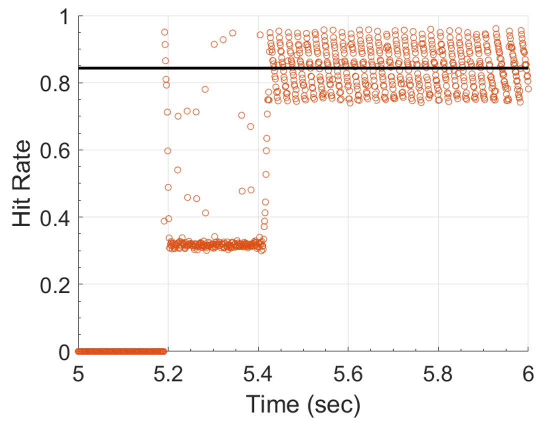
clients using the (shared) cache—or even the server.)

Of course, the entire goal of ActiveRMT is to enable multi-programming. Figure 3.6b 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 EWMA with $\alpha = 0.01$ against the left y axis; per-client throughput is plotted against the right y axis.

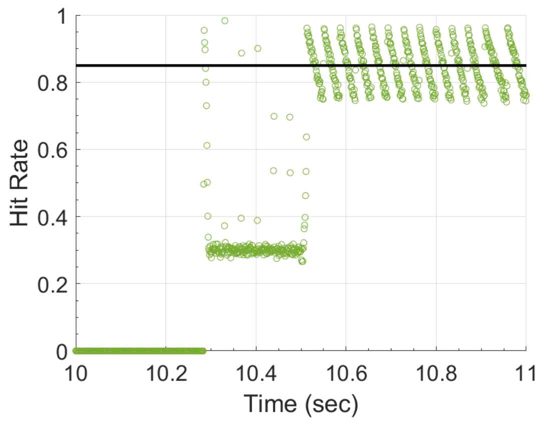
Each application instance, on arrival, triggers an allocation on the switch. The first three instances are able to take advantage of disjoint mutants (we use a most-constrained allocation policy to limit bandwidth inflation), thus obtaining exclusive memory regions (stages) and consequently zero disruption. 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)



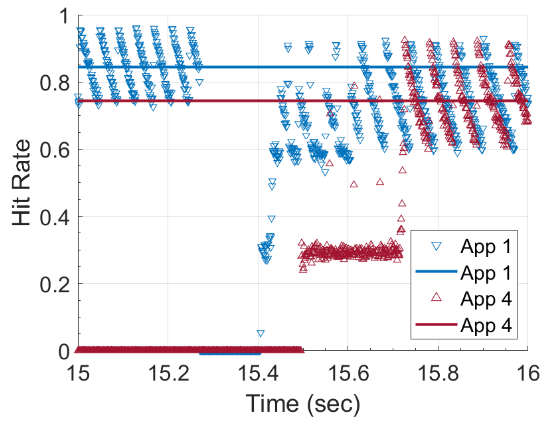
(a) Application 1



(b) Application 2



(c) Application 3



(d) Applications 1 & 4

Figure 3.7. Effective hit rate for each of the four application instances in Figure 3.4. Solid lines indicate the stable average hit rate.

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.

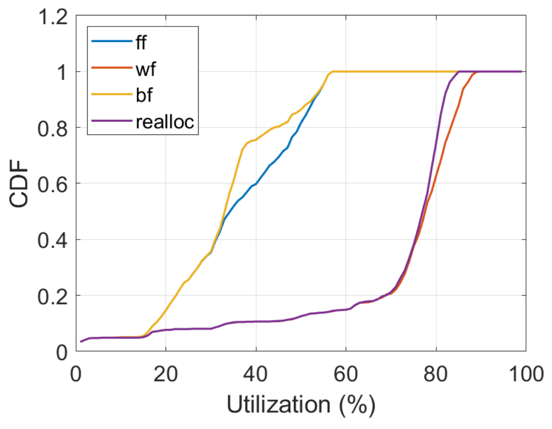
Allocations may disrupt active functionality of concurrent applications, causing all requests to be forwarded to the server in this example. Figure 3.7 illustrates, at finer time scales, the perceived hit rates of each of the application instances beginning with their respective arrivals. For the first three arrivals, we observe that the hit rate starts at zero and then eventually climbs up and stabilizes at $\approx$85%. (The intermediate $\approx$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 $\approx$ 150 ms disruption (at $T \approx 15.25$ seconds). During this time, the first application performs state extraction (Section 3.1.3) 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.

3.2.4 Allocation Alternatives

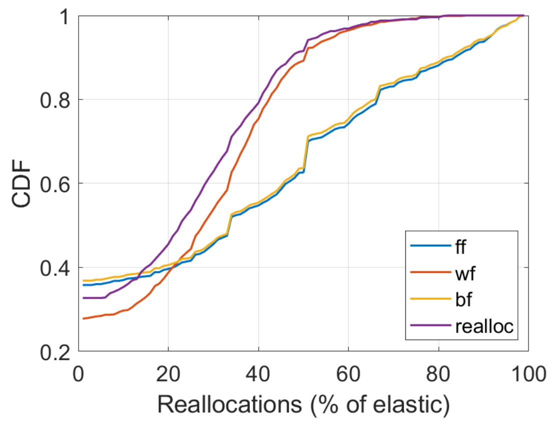
The previous experiments all employ a worst-fit allocation policy with a fixed granularity. Here we present an analysis of alternatives.

Allocation schemes

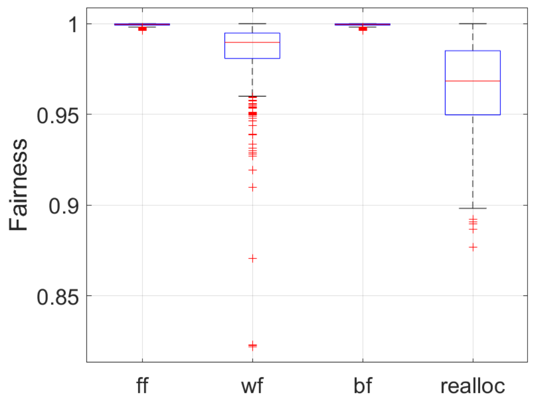
As described in Section 3.1.2 we choose a program mutant and corresponding stages based on the current occupancy of the stages. We employ a "worst-fit" (wf) allocation scheme that attempts to maximize resource utilization. Here we compare with other allocation schemes including "first fit" (ff) and "best fit" (bf). The former arbitrarily chooses any feasible allocation



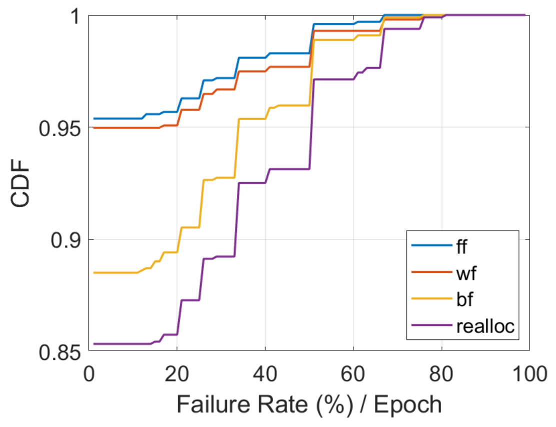
(a) Utilization



(b) Reallocations



(c) Fairness



(d) Failures

Figure 3.8. Comparison of first-fit (ff), best-fit (bf), worst-fit (wf) and reallocation-minimizing (realloc) schemes.

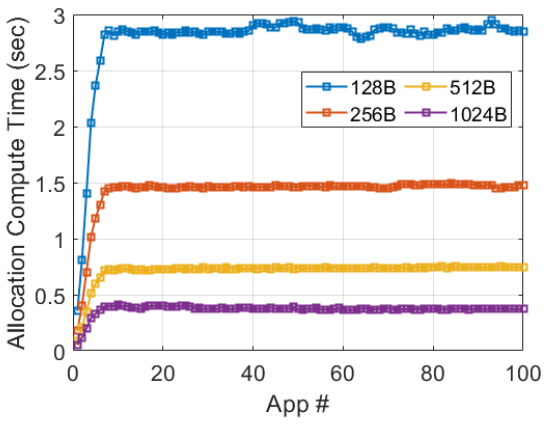
while the latter attempts to maximize per-stage occupancy. We also evaluate an allocation scheme that attempts to minimize the number of reallocations required to admit new applications (realloc). Figure 3.8 compares the allocation schemes over a simulated set of application arrivals. Consistent with 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

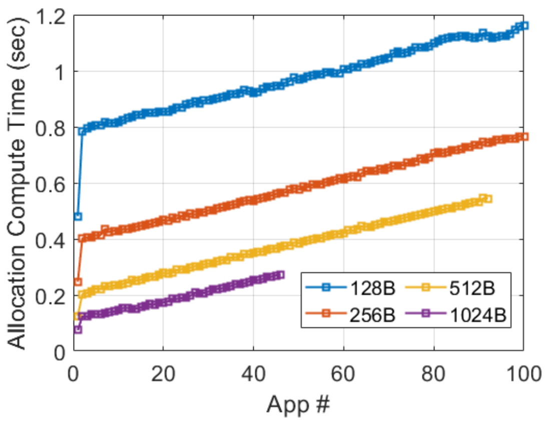
The amount of available memory in each stage is fixed for a particular instance of the runtime. However, the granularity with which the memory is allocated and, consequently, the number of memory blocks available in a given stage determines maximum occupancy and also impacts the time to perform allocation. The previous experiments use a granularity of 1-KB blocks. Figure 3.9 shows the control-plane allocation time for a sequence of 100 applications for four different application workloads with varying levels of allocation granularity using a most-constrained policy (c.f. Figure 3.2). (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.

3.3 Discussion

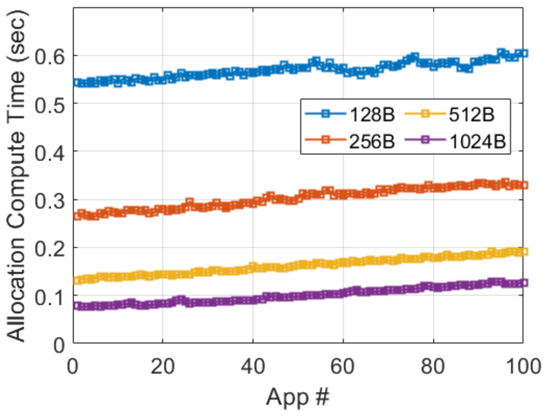
We present an approach to efficient dynamic resource allocations on ActiveRMT. By leveraging capsule-based active networking and program mutation, our prototype implementation is able to support dozens-to-hundreds of concurrent, stateful applications through efficient



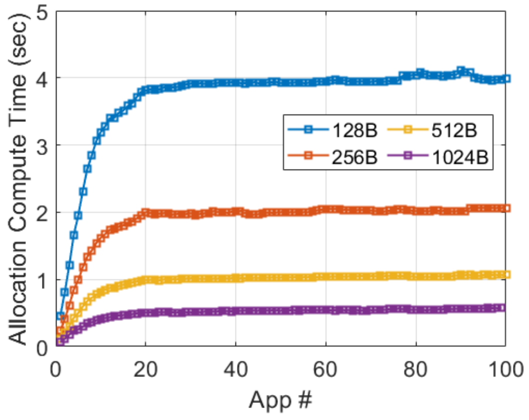
(a) Cache



(b) Heavy Hitter



(c) Load Balancer



(d) Mixed

Figure 3.9. Impact of allocation granularity on control-plane allocation time for four different application mixes.

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 may 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. For instance, while ActiveRMT provides tremendous flexibility in terms of specifying network behavior from an application's point of view, it is inapplicable in a non-cooperative scenario where behavioral inspection is required. Achieving both is however possible but requires a model for both trusted and untrusted users. It also requires revisiting approaches such as function chaining in such a context. We explore such a system in the next chapter.

Acknowledgements

Chapter 3, is a partial reprint of the material as it appears in – *Rajdeep Das and Alex C. Snoeren. "Memory Management in ActiveRMT: Towards Runtime-programmable Switches". In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. New York, NY, USA, September 2023, pp. 1043–1059.* The dissertation author was the primary investigator and author of this paper.

Chapter 4
vRMT

We believe that networks today are capable of running user-defined functionality at scale using emerging programmable switching hardware [12, 42, 91], and that such an ecosystem can be leveraged to offload application functionality as well as traditional network functions. While various frameworks for running network functions on general purpose servers have been proposed in the past [13, 26, 30, 41, 58, 61, 69, 70, 75, 82, 94, 95], very few [53, 99] have attempted something similar for emerging line-rate programmable switches. While the current ecosystem of programmable switches may fall short – in terms of memory and compute – in implementing all of the required functionality as of today, they are nonetheless effective in performing a restricted subset of tasks with high performance. This includes typical network functionality such as load balancing, routing, access control as well as application offloads such as key-value stores, telemetry, coordination and aggregation.

However, when abstracting devices such as programmable switches for generality several tradeoffs exist – such as being able to achieve a high degree of resource efficiency versus richer compute capabilities, or a highly restricted programming environment versus allowing trusted users. In this chapter, we consider the latter, which allows such a framework to support both administrator-defined functions that are able to inspect and control behavior of arbitrary network packets in addition to user-defined functions which are able to enhance performance and functionality of networked applications. This results in the following challenges – (1) automating function chain synthesis of heterogeneous programs, (2) classification of packets to function chains and (3) provisioning functions over programmable switches in a manner that makes fair and efficient use of resources. In this chapter, we explore these challenges, present solutions and evaluate network performance on a set of in-network applications targeted for such an ecosystem.

We thus present vRMT – a system for deploying application-specific functions that can be chained to co-exist with (operator-specified) network functions over a common network slice and in an automated fashion, thus enabling behavioral inspection over such runtime programmable networks. We correspondingly present a programmable packet processing model that can support operator-defined and application-specific functionalities over a network of authenticated users, that can support updates based on dynamic traffic patterns. Programs are defined in a *filter-process* specification which allows expressing functionality as lightweight as ACL policies to application offloads. Such a framework allows deployment of applications such as the widely used Maglev load balancer [22] and user-defined network policies onto a programmable switch.

While the benefits of a capsule-based approach may still be retained, our runtime programmable system based on RMT exports a programming model for both application developers and administrators alike that allow a precedence-based execution of programs chained together according to a filter composition algorithm. Subsequently, we present a dynamic line-rate packet classifier that assigns packets to function chains. To our knowledge, existing chaining techniques do not operate in a runtime programmable environment while existing runtime programmable environments do not support chaining, both of which are necessary to deploy network functions over programmable switches. Such a system also enables dynamic provisioning and migration of network functions based on dynamic traffic patterns. We explore one consequence of function chaining – packet recirculations – and how it can be managed by virtualizing switch backplane capacity.

4.1 Overview

Our goal is to enable deployment of a variety of in-network applications (and functions) over a programmable network using an access-delegated API from the network operator. We consider classes of network functions that perform behavioral inspection and alter default network behavior. This requires program behavior to be compliant with operator-specified policies. We refer to such functions as *privileged functions*. Network traffic that invoke such functions may contain active headers (Chapter 2) that carry application data or identify an explicit network slice (i.e. an FID). Such traffic is additionally required to be authenticated. The mode of function deployment over the network is not critical but is required to be secure and scalable. We adapt the language system of ActiveRMT to express programs.

We require a method of specifying functionality such that it can be composed programmatically at the controller. We use a *manifest* to this end. A *manifest* contains a program along with a memory demand for the program[1], a relative precedence of execution and a filter specifying the network slice over which to execute the program. Programmers interact with a controller (e.g. using gRPC) to deploy programs onto the network. The controller manages each switches' resources and provisions functionality across the software-defined network through switch control plane APIs.

Traffic corresponding to certain application offloads – particularly those that contain application data – use the same form of encapsulation as ActiveRMT (except the programs). To prevent forgery and tampering, active packet headers contain an *access token* (Section 4.1.2) and a *message authentication code*. These packets typically correspond to application offloads. Operator-defined functions can be invoked on regular IP packets and do not require any form of encapsulation, enabling in-network functionality that is able to inspect the behavior of other packets (e.g. firewalls). Application data is carried within active headers. Default

[1]In contrast, ActiveRMT (Chapter 3) did not require the controller to be aware of programs but simply their memory access patterns. Such a design was less restrictive in the sense that it allowed arbitrary programs to (re)use the same allocated memory regions.

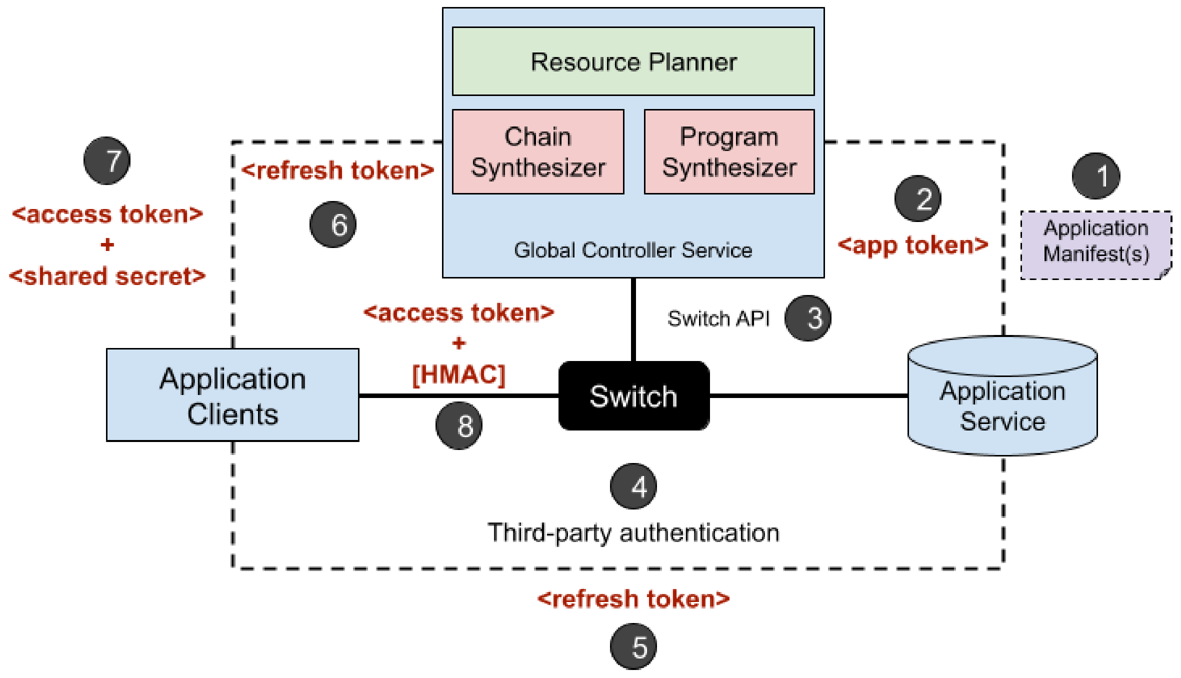


Figure 4.1. A model for deploying network functions onto a vRMT network. A global controller performs the task of provisioning in-network applications onto a programmable switch.

network operation forwards traffic according to operator-defined rules. Application services communicate with the controller over an encrypted path (we use gRPCs in our implementation), while application clients interact with the network over an unencrypted data path.

4.1.1 Deployment

Figure 4.1 illustrates a system model for deploying network functions over a vRMT network (e.g. a rack). The controller provisions an application function by first allocating resources for the program on switch hardware and subsequently synthesizing classifier rules to identify relevant packets. The following steps are performed before packets transiting the network can invoke user-defined behavior: (1) Applications specify functionality using a manifest, which contains a program along with its memory demands and a corresponding policy for executing the program. (2) It then generates an *application token* which can be used to identify and authenticate the application with the controller in future. (3) The controller then provisions the application onto the relevant switches. The subsequent steps apply to clients that embed application data within their packets (or carry the active header). (4) Application clients authenticate with the application service using proprietary or third-party authentication schemes. (5) The application then provides the client with a unique *refresh token* that can be used to authenticate the client with the controller, effectively authorizing the application to invoke active functions over the data plane. (6) The client communicates with the controller using this *refresh token* (7) to obtain an *access token* and a *shared secret*. (8) The *access token* is embedded in the packet headers and the secret is used to compute the HMAC.

The manifests contain a traffic class specifying which packets to invoke the functionality on. Programs may contain static data (e.g. IP addresses for an ACL), which are used for matching on variables or setting the value of variables. Such data is passed along with the program at the time of deployment.

Static program data is installed along with match-action table entries (thus counting towards the size of the program). The toolchain for composing and compiling programs is split across the applications and the controller (Figure 4.2). Programmers write functionality in a high-level language or equivalent instructions (Section 2.4). The corresponding instructions along with accompanying data and a packet filter expression is composed into a manifest. The controller compiles this program for each relevant switch while taking into consideration switch-specific constraints and existing resource allocations.

4.1.2 Authorization

For applications that require explicitly specifying in-network functionality, packets encapsulate information identifying functionality to be invoked on the switch. Such active traffic (i.e. traffic containing active headers) is required to be authenticated. Application offloads (e.g. in-network cache, in-network aggregation, active probing, etc.) fall into this category. Such application functions require client coordination prior to using the activated network. Clients authenticate themselves with the application service outside of vRMT (e.g. using a third-party authentication scheme). Once authenticated, the application service assigns a *refresh token* to the

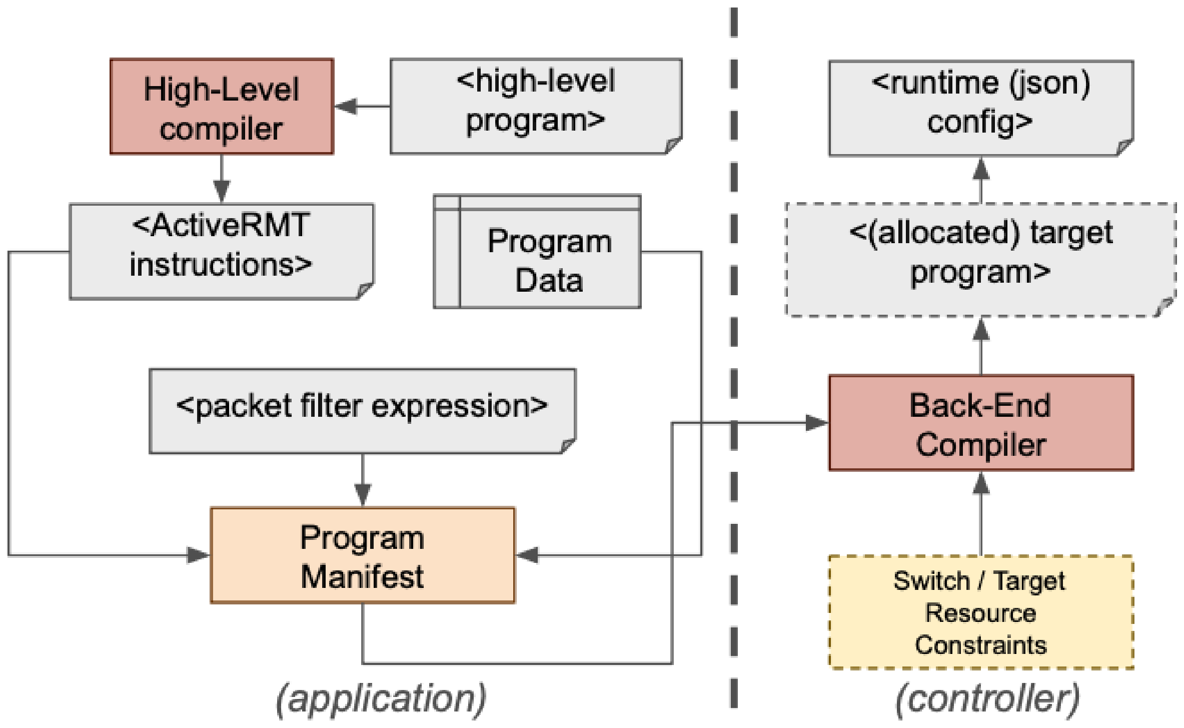


Figure 4.2. Programming toolchain for user-defined applications. The toolchain is split across applications and the controller – vRMT instructions are generated and the program manifest is composed by the applications. The controller synthesizes the final program based on switch-specific resource constraints. This model allows achieving target-independence of application programs in vRMT.

client. This refresh token is also sent to a vRMT authentication (introspection) endpoint, thereby authorizing the client on vRMT. Clients communicate with the authentication endpoint via a secure channel (e.g. SSL) to obtain an *access token* and a *shared secret* using its refresh token. These access tokens are embedded in client packets as a field in the vRMT header to authenticate the packets on a vRMT switch and invoke the corresponding function.

Access tokens are unique to an application – all application clients use the same access token at a given point in time. We choose to do this since all clients access the same resources. Alternately, using unique tokens for each client does not necessarily make the system more secure, since all clients have access to the same resources. However, compromising the access token at one client's end compromises the entire application. The frequent refreshing of tokens reduces this window of vulnerability.

Access for a particular client can be revoked by invalidating its refresh token. This cuts off client communication with the authentication endpoint and the client is unable to acquire new access tokens. Note however, that the client is still in possession of the current access token can access the services for the duration of the refresh interval. Various approaches can be adopted to push updated access tokens to authorized clients post such an event, as an alternative to periodic refreshing of access tokens.

4.1.3 Protection

Each function is assigned a memory region and a network slice for operation [2]. Such a sandbox constitutes a protection domain for the function. Function identifiers (FIDs) essentially behave as domain protection keys. Using domain protection keys for both instruction and memory enables operators to restrict functionality to only trusted users. An example of such a scenario would be the use of a `SET_DST` instruction that sets the destination IP address of a packet. Load balancers for example, could use this instruction to set the destination IP (instead of tunneling). However, such an action could also be abused to implement a superspreader [81] – an operator may wish to restrict such types of functionality. However, since programs need to be provisioned onto the switches before they can be invoked, such programs can be checked against respective protection domains prior to provisioning.

Memory protection domains can be inherited allowing child processes to access the memory location of their parent[3]. Applications in vRMT are allocated memory only once. This prevents multiple application functions from hoarding memory with multiple allocations. However, an application may require alternate functionality to be invoked over the allocated memory region. We thus enable memory sharing among application functions. For example, in the case of our Maglev application, the load balancer application function can perform load

[2]In contrast, ActiveRMT applications are only assigned a memory region which are identified by a *FID*. Application traffic is implicitly sliced based on their identifiers.

[3]In ActiveRMT, this was not necessary since application clients could invoke any program on the switch which could access the allocated memory region.

balancing while another health checker can update the backend pool. Both these functions require access to the same memory region – the backend pool.

4.2 Function Chaining

A virtualized network function processing device is expected to have function chains (or service chains). While such function chains typically reside on x86 processors, hosting them on a programmable switch presents several challenges. Typically, service chains are specified by the network operator [69, 70] whereas a system that supports dynamic function loading from multiple users requires some method of automated chain synthesis (based on operator-defined policies). Chain execution is typically associated with schedulers [55, 58, 69, 82] that move packets from one network function process to another (potentially on the same device). Alternative approaches synthesize chains from multiple functions [48, 70] to avoid scheduling overheads. Such approaches are not necessarily favorable on programmable switches, which have much more stringent resource constraints. Approaches that emphasize reusability are more efficient and desirable for such targets. Leveraging the capabilities of RMT hardware, we adopt a hybrid approach where chain dispatching and function scheduling is performed on the switch data plane while the task of determining a static function chain is delegated to a control plane classifier – a data plane module that assigns a function chain to a packet upon arrival is configured by a control plane module that determines such a configuration. Determining such a configuration for the switch data plane is equivalent to a hardware packet classification problem.

4.2.1 Authentication

The vRMT network does not require packets to carry any special headers for normal operation. However, application-defined functions could require packets to carry application data. We use access tokens to identify application functions for such packets. As a proof-of-concept, in order to preserve integrity of such data, we use message authentication codes (HMACs). HMACs are computed over the access token, function arguments and a shared secret. HMACs can be verified at line-rate on programmable switches (at a certain overhead). In our implementation, a 64-bit key is used to hash the packet data fields. The CRC32 hash function is used to compute MACs on a Tofino switch. While considered cryptographically insecure, this can easily be replaced by a more secure hashing unit on future devices.

On a Tofino device, packets pass through parsing and match-action stages before being queued by the traffic manager. The switch can accept packets at line rate and feed them to the ingress pipeline. However, malicious traffic may attempt a denial of service (to both network functions and regular traffic) by overflowing queues. To prevent this, traffic must be authenticated and authentication must be performed within the ingress pipeline prior to queuing. We perform authentication prior to packet classification in our design.

4.2.2 Packet Classification

Provisioning a heterogeneous set of programs, each with their own respective filters leads to a multi-match classification problem. A packet could trigger multiple functions which must be executed according to some order prior to forwarding the packet. Such a set of functions would constitute a function chain. Inferring such a chain at runtime is impractical on current RMT switching hardware. An online algorithm synthesizes chains and respective classifiers on program arrival based on filter specifications and updates the switch accordingly.

A BPF-style filter is used to specify which packets the corresponding program should be invoked on, which compiles to a set of table configurations. The operators convert the respective operands to range values. We choose fields that are used in standard header matches along with an explicit function identifier (FID) to help us identify traffic that is designated for a network function. A header in this space corresponds to a (5+1)-tuple *(ip_src, ip_dst, ip_proto, udp_src, udp_dst, tcp_src, tcp_dst, FID)*. Every application-defined in-network service operates on a network slice which is subset of this space. Due to line-rate processing constraints, we are limited to using hardware classification techniques (e.g. using TCAMs). An example filter expression is show below – this expression filters all *HTTP* traffic that arrives from *192.168.0.0/24* and destined towards *10.0.2.0/24* or all *TCP* traffic encapsulated with active headers identifying function with *FID=1* from *192.168.1.0/24* and destined towards *10.0.3.0/24*.

```
(ip_src == 192.168.0.0/24 and ip_dst == 10.0.2.0/24 and
   ip_proto == 6 and tcp_dst == 80)
or
(ip_src == 192.168.1.0/24 and ip_dst == 10.0.3.0/24 and
   fid == 1 and ip_proto == 6)
```

Challenges

Various approaches to packet classification have been proposed in the past [35, 34, 54, 56, 80, 92]. However, most solutions cannot be implemented on a device such as Tofino. While the flexibility of P4 and RMT allow user-defined classifiers to be implemented on such a device, we are constrained by switch resources when attempting to integrate a classifier. We focus on schemes that are both sufficient and efficient given our use case – a set of filters associated with network functions.

A strawman approach to packet classification would require the use of geometric intersections [92]. We could define a single-stage classifier containing all the fields (dimensions). This approach has several problems. Rules contain ranges for each field and pose a problem – a set of ranges cannot be ordered. This would require encoding overlapping regions, which can grow exponentially – although the number of functions and hence rules are in the few 10s and hence not a problem. However, this is infeasible on current hardware – a table can contain at most one LPM match key; Moreover, LPM matches cannot be estimated using range matches since these keys are limited (e.g. on Tofino) to 20 bits. Hence, keys have to be split among multiple tables. The smallest set of such tables would require the two LPM matches to be split across two distinct tables. The rest of the fields can be assigned to either.

Computing function chains can be achieved by performing a combinatorial expansion of the set of functions and combining them using conjunctions. The corresponding BPF expressions is then converted to a canonicalized disjunctive normal form, where each minterm contains all the fields. Each minterm can then be interpreted as a distinct table rule. Rules can be ordered in TCAM-compatible order [92] based on the length of the chain – longest chains precede shorter chains. This approach has exponential complexity in the number of functions, but could work well in our scenario since we expect only a few 10s of application functions to be provisioned onto a single switch. However, better approaches exist that make most efficient use of switch memory.

Existing multi-match classification schemes [56, 80, 92] require hardware support to be implemented efficiently, that is absent on current RMT devices. For example, to realize TCAM compatible ordering one could use an indirection table for set intersections of the classification rules. However, match tables on RMT devices can only retrieve one action per stage – such an indirection would require multiple and variable number of stages and is hence impractical to generalize. Our approach avoids such ordering altogether by using disjoint partitions on each field. The use of cross-products, although costly, is however inevitable in our context where we are limited in the number of RMT stages that can be used for packet classification. We use a combination of SRAMs and TCAMs to alleviate the costs.

Classifier Design

We use a two-phase classifier to determine a function chain to execute upon packet arrival. We use extracted internet 5-tuple (and active header encapsulated) fields from the parser to perform the classification. The first phase of the classifier assigns a subclass to each field. We choose a disjoint partition set [49] to encode each field in the classifier. The corresponding match tables use longest-prefix match (LPM) and range matches for the address and non-address fields respectively. We expect the number of entries for such fields to be small in practice (Figure 4.3) and hence not expensive to use. A partition is defined as a subset of the header

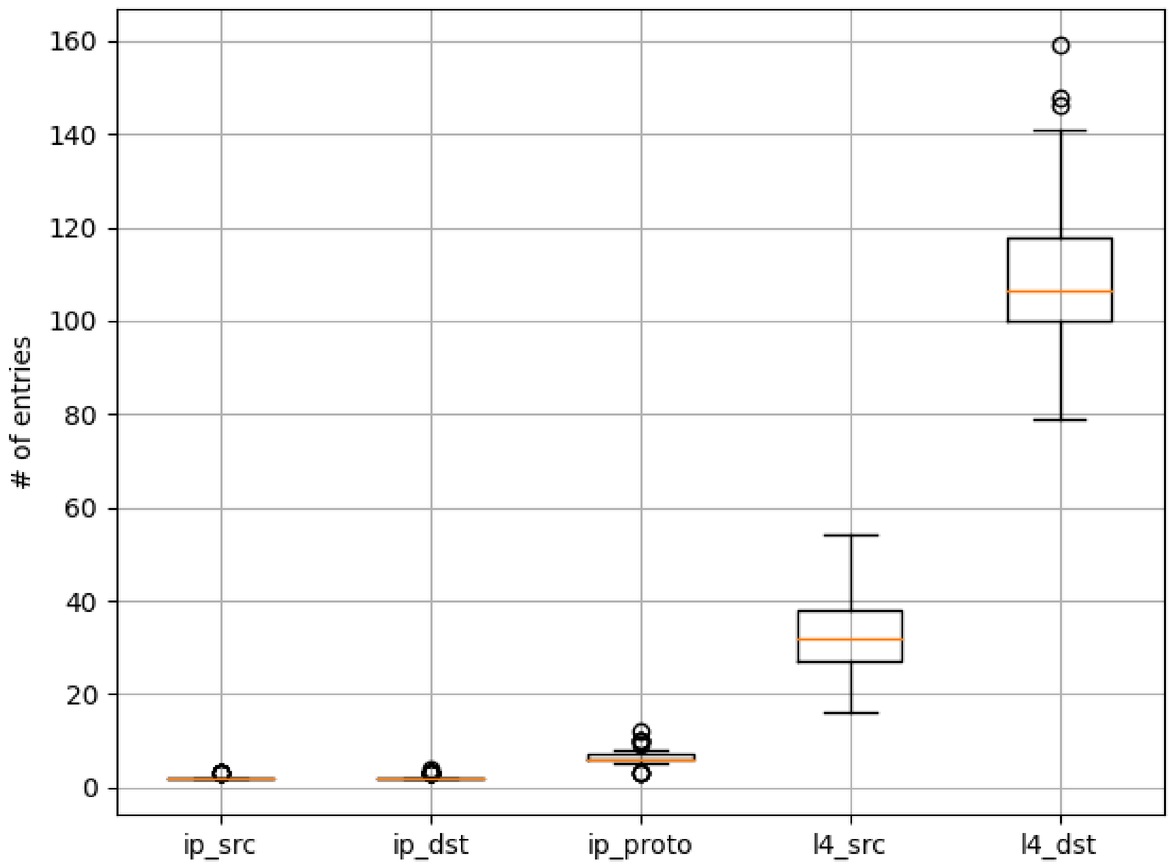


Figure 4.3. Number of entries required for each field in the classifier for a randomly sampled subset of 500 rules from a Snort database. An arbitrary IP configuration was chosen for this experiment.

space corresponding to that field. All partitions in the classifier are mutually exclusive (but not necessarily contiguous). For overlapping rules that are not a subset of the other, we split the rules into disjoint partitions and create a separate rule for each partition. A set of partitions form a sub-classifier for a field. The second phase of the classifier assigns a chain identifier based on the subclasses assigned to each field. This table effectively behaves as a dispatcher for the function chain. Relevant tuples in this classifier are inferred based on cross products. Since such a table can easily grow in size, we use an exact match table using SRAMs.

Figure 4.4 illustrates our classifier and the process in which a chain is assigned to a packet. The IP source and destination fields use LPM matches while the layer 4 (UDP/TCP ports) source and destination fields, and the IP protocol fields use range matches. Each field is contained within its own table. The field tables are matched first and in parallel and a respective set of IDs is obtained. If the packet is an active packet, then concurrently the active header is also authenticated. First, the shared secret, FID and expiry time is retrieved via a table match on the access token. The extracted access token, application data and secret are then used to compute a HMAC. This is then compared with the HMAC embedded within the active header to validate the packet (the expiry time is also validated). The FID and field IDs are then matched on a dispatcher table which assigns the packet a function chain (CID).

As an optimization, functions belonging to a single application may be composed into a single one. However, this should not be generalized to the entire set of functions residing on the switch; Not only does this result in consumption of more (limited) switch memory due to redundancy, it also requires state replication for multiple instances of the same function residing on the same pipeline.

Chain Synthesis

Our objective is to ensure that a packet can match on at most one entry in all tables, thus obviating the need for rule ordering – a condition that is necessary to enable runtime updates to the classifier. The number of distinct function chains required to be provisioned on the switch

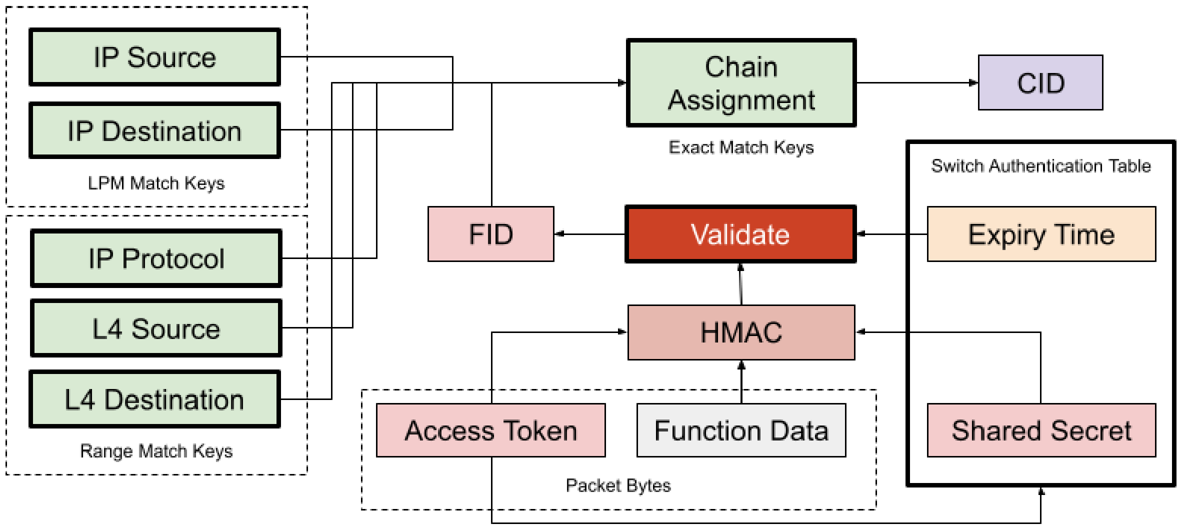


Figure 4.4. Assigning a chain identifier (CID) to a packet based on headers and data. For packets that do not contain access tokens, a default FID (0) is used in the classification scheme.

depend on the set of filter rules corresponding to the provisioned functions. This set of chains are updated online upon addition of each new rule, and the classifier correspondingly updated. While approaches to perform chain synthesis based on mutually distinct packet filters [49] have been proposed in the past, we do not know of any prior approaches to implementing an online classifier for a dynamic set of network functions. We describe our online classification algorithm next.

When adding a rule, we insert each field's value into its corresponding sub-classifier. Each element in the field partitions is a contiguous range of values in their respective domains (e.g. IP addresses, positive integers). Apart from the IP addresses, the rest of the fields can be compared using range matches. IP addresses use LPM matches and hence the respective tables do not need to be partitioned explicitly. Each element in the set is associated with a range (start,end) specifying the partition and a set of labels identifying the function that is relevant to that partition – IP addresses use LPM values. Each set of such partitions is ordered by their range start index. When a new partition is inserted into the set, we perform the following. If the element's partition coincides with the new partition, then the function identifier is added to its corresponding set of labels. Otherwise, if the partition is completely disjoint with every other partition then it is inserted as a new partition in the respective location and the corresponding identifier is updated. Else if there is an overlap with an existing partition, a set of subpartitions are formed accordingly and the corresponding labels are updated. This ensures that partitions are non-overlapping. During this phase, the set of affected partitions for the inserted rule are retrieved for each field. A cross product is then formed with the affected partitions across all fields and inserted into the final classification table.

Ordering

Typically, precedence for functions within a chain is defined by a network operator [29, 65, 69]. Our approach is modular, where a function chain is synthesized by the controller. For functions that do not modify packet header data subsequently used in other functions, this is not a problem. Our model allows mangling of only a limited set of fields (e.g. destination IP address). In addition, application data arguments used in active headers can be modified by functions. The ordering of functions is thus dependent on the chain itself. Functions that require behavioral inspection and depend on fields that cannot be mangled (e.g. TCP flags) should be inserted at the beginning of the chain for efficiency. Functions that depend on mangled fields should be placed at the end of the chain.

For functions that are defined by the same user, ordering can be specified by the user (we use priorities in our implementation). Such functions cannot be used to inspect the behavior of other packets and hence should be inserted somewhere in the middle. We choose a policy where operator-defined functions have higher priority than application-defined functions. Such a policy allows security functions to have precedence and prevent further execution of malicious packets.

4.2.3 Classification Performance

We evaluate whether our classifier can be realized on a Tofino device and the scalability of the system that can be achieved with such a scheme. In order to maximize the number of processing stages, we attempt to minimize the number of stages that we can use to implement the classification tables and the amount of TCAM and SRAM memory on the switch. We thus build a two-stage classifier to dispatch function chains. Our classifier can hold configurations corresponding to 100s of functions.

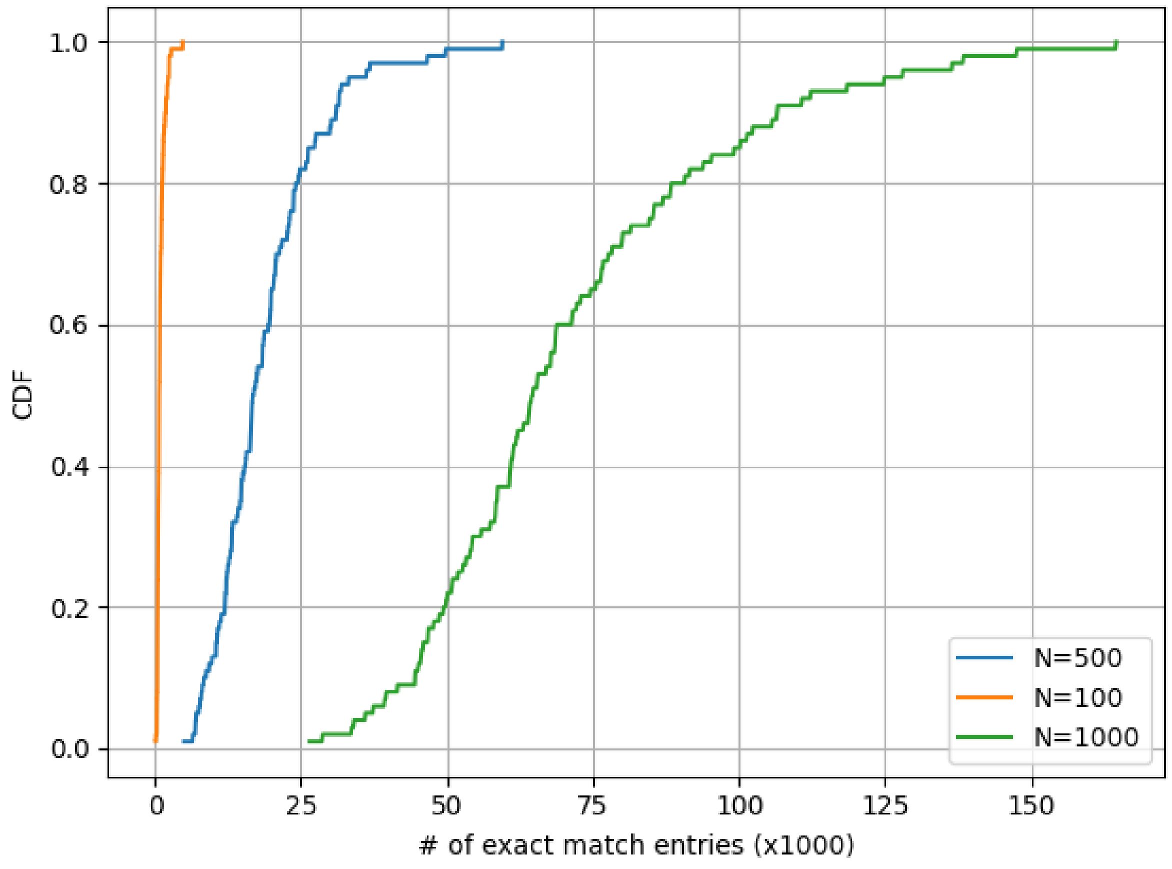


Figure 4.5. Number of exact match entries used plotted as a CDF across multiple sets of randomly drawn samples of rules from a Snort database. Size of the exact match table is 32k entries.

We evaluate our classifier to determine whether given a certain number of filter rules that represent a reasonable number of functions (a few 10s), can the classifier accommodate those functions. To that end, we use a Snort [16] database to sample rules and populate the classifier online. These rules potentially represent a function's filter expression (based on commonly used ports, etc. in current network applications). This database contains ≈44k header match rules. We choose header matches corresponding to internet 5-tuples and synthesize packet filter expressions for them. We then insert these rules into the classifier data structures and determine feasibility by inserting them into the switch. The final chain assignment table can hold 32k entries. Figure 4.5 shows the expected chain assignment table usage for a set of 100/500/1000 rules uniformly sampled from a Snort database. We add each rule one at a time to mimic an online classifier. This experiment is repeated 100 times. For each rule in a set, we computed the field partitions (new and modified), computed the cross products (new and modified) and inserted the corresponding entries into their respective tables. The actual number of table entries required is feasible on switch memory for over 500 rules (where the exact limit depends on the sampled set of rules). Figure 4.6 shows that the number of actual entries required is much less than the worst-case based on cartesian products.

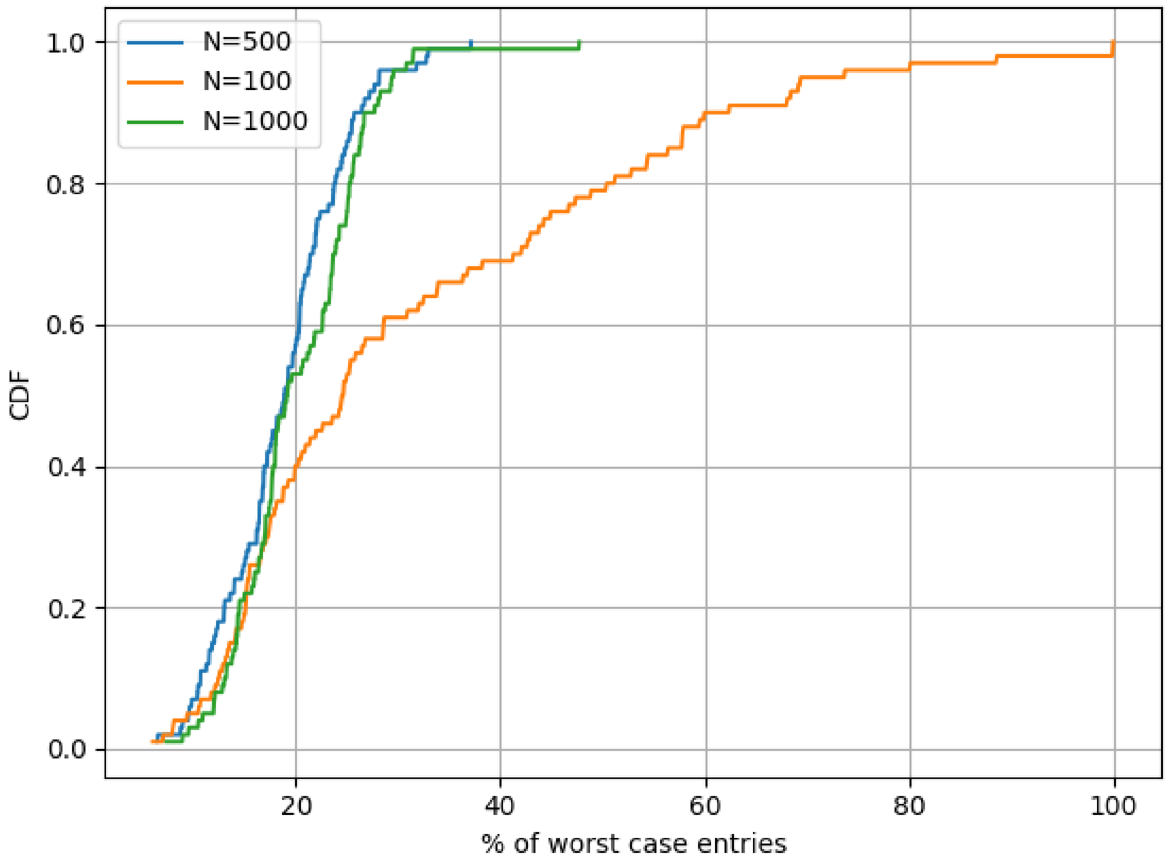


Figure 4.6. Percentage of actual entries used in the classifier as compared to the maximum number of cross products that could be formed using the respective field entries.

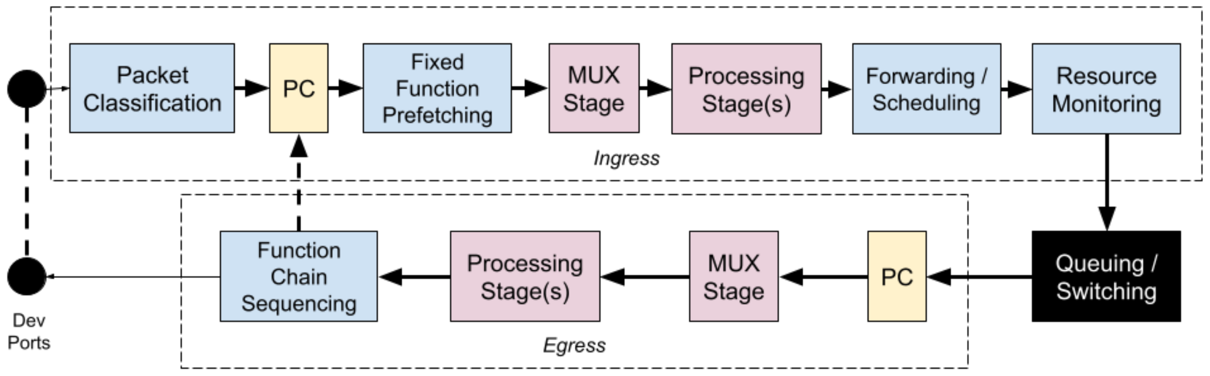


Figure 4.7. vRMT architecture overlayed on top of a PISA pipeline – consisting of a pair of ingres/egress match-action pipelines, a traffic manager and parsers (not shown here).

4.3 Switch Runtime

vRMT's instruction processing model differs from ActiveRMT, which prioritizes resource efficiency on RMT pipelines. While a capsule-based approach can be adopted in such a scenario, one needs to extract a segment of the program from the packet in each iteration, verify the program and then execute it. A more efficient approach is to store the programs on the switch instead. vRMT adopts this approach (it also adds more programming capabilities to the virtualized architecture). Instead of parsing program segments from the packet, program instructions are stored across the tables and state tracked by a program counter. An instruction is stored on each table as a *(iteration,FID)* pair and a program counter is used to maintain program state. An instruction table in vRMTmatches on the program's function identifier, control flow state, program counter and the contents of a pair of multi-use registers. These registers enable – in addition to memory isolation and predication – exact and range table matches on program data. This subsequently allows the implementation of certain data structures in a high-level language such as sets and dictionaries. Such a capability is enabled by a control-plane authenticated deployment mechanism (as described in Section 4.1), which allows users to specify both code and data.

Figure 4.7 illustrates the vRMT architecture imposed on top of a PISA packet processing pipeline. As with ActiveRMT the parser (not shown in the diagram) is not programmable; Instead the parser recognizes TCP/IP headers and the general purpose active header. Parsed packets are first identified by the classifier as belonging to one of the hosted functions or otherwise. Consequently, the stored function is executed on the packet and forwarded accordingly. Packets that are not handled by any of the switch programs are forwarded according to IP forwarding rules. Function execution is a multi-stage, multi-phase and iterative process that generalizes computation at the expense of backplane bandwidth.

4.3.1 Domain Switching

There are several ways to implement function chains such as dynamic scheduling [58, 69] and synthesis [49, 48]. Following prior work, we could synthesize all the functions in a chain into one monolithic function and provision it onto the switch. However, this would be inefficient in terms of table entries, since a function would need to be duplicated (starting from a distinct stage) for each chain containing the function. This would not only need more instruction table entries, but also more stateful program memory. In addition, this may require synchronizing state across all memory regions for the same function – something that is impractical to achieve at for high-throughput devices. Instead, we choose to modularize chains – there is only one instance of a function on the switch and all chains that contain the function simply point to it. Such a design also allows function chains can be updated (hitlessly) at runtime.

A function chain is dispatched once a packet is assigned a chain identifier (CID) upon classification. This identifier determines the sequence of functions that are scheduled to be executed on the packet. An initial function identifier (FID) is also assigned post classification. We use a state machine to schedule functions in a chain. A chaining table implements this state machine, which maps a CID and current FID to the next FID. This lookup table essentially performs a domain switch from one function to another. The packet header vector (PHV) is updated with the new protection key (FID), which invokes the corresponding instructions at the respective stages. This switch is performed at the end of the egress pipeline. A function may terminate early, causing a domain switch to happen earlier and thus reducing the number of recirculations required to execute the function chain.

Packet drops during recirculations may result in incomplete execution of function chains. This may lead to incorrect application state. We leave it to the task of compilers to ensure that programs are generated in a way that prevents inconsistent state due to incomplete execution.

4.3.2 Program Execution

We structure a RMT pipeline with preset tables that implement our general-purpose packet processing model. Post classification, a packet is assigned a chain identifier and an initial function identifier. Programs are executed in "passes" where one pass covers all ingress and egress processing stages. Similar to ActiveRMT, a function requires a minimum of one pass – this is done to allow modularity when synthesizing function chains. Additional passes naturally require packet recirculation – a technique that we normalize and quantify in our design. Passes are tracked using a program counter (PC); As an optimization, this counter is incremented at both ingress and egress – chains that complete execution at the ingress can bypass egress to save latency. This approach allows arbitrarily long programs to be run on the switches. A series of processing stages then execute the program instructions. The packet destination is then decided at a forwarding table and execution is then transferred to the egress. As another optimization, certain fixed functions that require scarce hardware resources (and whose output do not change for a packet) are prefetched and stored in the PHV for later use, such as random numbers and flow hashes. These values are either used only once or do not change during the lifetime of the packet throughout the switch. Obtaining these values later on consumes resources such as hash units which can otherwise be used for other user-defined operations in the processing stages.

Certain types of functionality such as in-network telemetry and being able to express Snort rules requires reading header fields such as TCP options. Due to the limited size of the PHV container groups, a very few of them can be accommodated limiting the scope of functionality. vRMT's processing stages are thus divided into two categories – multiplexing stages and regular processing stages. The former is included to address a limitation on RMT pipelines – data movement and arithmetic-logic operations can only be performed among PHV containers residing within a container group. This limits the number of PHV containers that can be used for processing a packet. To perform computation over containers located across container groups one needs to move data from the source container group to a destination container group and perform the computation there. Fortunately, such a feat can be performed using an identity hash (a hash algorithm using a custom function that effectively copies the original value to the destination), by hashing an object from one container group to another. We leverage this trick to devise a multiplexing stage which moves data from an alternative container group to the main container group. We unfortunately cannot duplicate the capabilities of the regular processing stage onto the multiplexing stage since that leads to a write-conflict. We thus harness the multiplexing stage to strictly perform data copy from PHV fields (including switch-specific metadata fields). A PHV container is used as a buffer in the multiplexing stage where data is copied from other fields. The contents of this buffer is then identity hashed into a processing buffer in the main container group, where it can be used by the regular processing stages.

4.3.3 Hitless Provisioning

The sequential nature of RMT pipelines allows us to perform updates that appear hitless. The program instruction tables for the new FID are updated first. Based on the compiled sequence of instructions and the total number of active processing stages, we calculate program counter values and insert the instructions into their respective stages. The ordering of operations is necessary since the classifier can assign packets to existing chains while the switch is being updated.

We then update the chain state transition table. This table has entries of the form *(chain_id, current_fid)* $\rightarrow$ *next_fid*. For new chains, the entries for each state are updated in any order. For existing chains, the insertion order of entries depend on where they are inserted in the chain. For insertions at the end, the last entry of the chain is updated to reflect the new FID. For insertions in the middle and at the beginning, the transition from the new FID is inserted first followed by updating the previous entry to point to the new one. If the insertion is at the beginning of the chain then the corresponding chain assignment table is updated.

The classifier is updated next. The chain assignment table is first updated with the new cross-product tuples. We update the table with the modifications first, followed by inserting the new tuples (although this restriction may be relaxed in certain scenarios). This way packets that match on existing field tables would be processed correctly (e.g. inserting an ACL). The field tables are then updated with the range modifications first followed by the new partitions. This set of updates provisions the new function along with its rule.

There are three sets of match-tables that need to be updated when the classifier is updated – the TCAM-based field tables, the SRAM-based chain assignment table and the chain state transition table. A rule is first inserted into the partition set sub-classifier which resides on the controller. This classifier outputs a set of new partitions (with their identifiers), the set of partition identifiers that correspond to the new rule and the modified partitions (which were shrunk in size). This is then fed to the chain assignment sub-classifier that outputs a set of new chains, new cross-product tuples and modifications of existing tuples.

4.3.4 Implementation

We implement our runtime using P4-16. The controller is written in Python. We have written a high-level language compiler for imperative vRMT programs using Python and an associated tool to generate a manifest. To measure switch counters frequently (e.g. to obtain recirculation bandwidth usage), we wrote a vanilla C++ wrapper for the switch driver which periodically reads switch counters using Tofino C++ APIs. The read counts are communicated to the controller via a Unix Domain Socket (UDS). The controller uses these counts to perform measurements. We use a 100 ms polling interval based on the granularity of the switch traffic counters. The lag associated with each measurement epoch is $\approx$10 ms. We perform experiments over two x86 servers with 20 cores connected to a Tofino switch over 40G links.

As a part of our evaluations, we use a set of three applications in-network telemetry

Table 4.1. List of vRMT functions written using our domain specific language.

#	Function	Description	Recirculations
1	ACL	Access control list. Checks if a packet belongs to a blacklist and then drops it correspondingly.	0
2	CMS	Count-min-sketch. Performs a count-min-sketch based on a 5-tuple.	2-3
3	MAGLEV	Maglev load balancer. Load balances according to the Maglev algorithm.	1-2
4	SYNFLOOD	SYN flood prevention tool. Prevents SYN floods using rate limiting.	0-1
5	TELEMETRY	In-network telemetry. Fetch congestion information from the switch be measuring the delay from the ingress to egress pipeline. Accumulates the number of packets congested according to a threshold.	1
6	SFW	Stateful firewall with connection tracking. Allow connections to be initiated only from an allowed network segment.	3

(INT), TCP SYN rate limiter (SYN) and the Maglev load balancer (LB). We choose these three applications to be a representative of typical services that are likely to be deployed within a service chain. INT measures congestion on the switch and stores statistics in switch memory. SYN rate limits SYN packets from TCP flows to prevent SYN-flood attacks. LB performs load balancing based on the Maglev algorithm. Due to the nature of Maglev hashing, such a load balancer is an ideal candidate for programmable switches with limited memory. We choose a table size which is one less than a Mersenne prime (e.g. 64k), to have minimal impact on load balancing. DSR is achieved using a separate program implemented as a gateway on a vRMT switch. Since vRMT updates IP checksums on the switch, we omit tunneling in our implementation.

4.3.5 Active Programs

We have implemented several in-network applications to run on vRMT. Table 4.1 lists some of those applications. While mostly similar to ActiveRMT, vRMT allows programs to operate on certain additional fields and exhibit additional behavior (e.g. setting the destination IP address), allowing richer functionality such as access control. One caveat however, is that vRMT has 12 processing stages as compared to ActiveRMT, due to overheads of authentication, classification and chaining. Nevertheless, a number of useful applications can be implemented with a reasonable number of recirculations. Following are descriptions of some of the applications.

Stateful Firewall

We demonstrate the utility of vRMT using an example firewall application. Listing 4.1 shows a firewall function that allows TCP connections to be only initiated from a network segment. Two stateful objects are used here – one corresponding to the direction of traffic (i.e. from the LAN or the WAN) and the other corresponding the the connection table. The program performs the following: if a packet carrying a TCP SYN flag was initiated from a LAN port (specified in the `direction` register), the firewall stores the corresponding flow (on the `source` register identified by the 5-tuple hash and source address) and allows all subsequent packets. Packets originating from outside the network segment are blocked.

In general, vRMT allows achieving functionality similar to IPTables on the switch. The traffic classifier (as described in Section 4.2.2) allows packets to be filtered based on a set of fixed function header fields. Additional filtering can be performed as described in the examples here. Such a model extends typical IPTables chains to a programmable switch.

SYN Rate Limiter

As another example, we demonstrate a TCP SYN rate limiter, an application commonly used in networks. Such a program can be adapted to rate-limit other types of packets as well. Listing 4.2 shows the active program for such an application.

This program uses one stateful object, `last_ts`, which stores the ingress timestamp of the last observed SYN packet. We choose to rate-limit at the granularity of a second and hence read a coarse timestamp from the switch – since timestamps on Tofino are 48-bits wide while the vRMT word size is 32-bits, we cannot load the entire timestamp, leading us to split the

```
GET_PORT
MAR_ADD_MBR
MEM_READ,#direction
CJUMP,@LOC_1
LOAD_TCP_FLAGS
CONST_LOAD_MBR2,$(2)
BIT_AND_MBR_MBR2
CJUMP,@LOC_2
LOAD_TCP_5TUPLE
COPY_MAR_MBR
ADDR_MASK
ADDR_OFFSET
LOAD_IP_SRC
MEM_WRITE,#source
RETURN,:LOC_2
NOP,:LOC_1
LOAD_TCP_5TUPLE
COPY_MAR_MBR
ADDR_MASK
ADDR_OFFSET
MEM_READ,#source
COPY_MBR2_MBR
LOAD_IP_SRC
MBR_EQUALS_MBR2
CJUMP,@LOC_3
DROP,:LOC_3
RETURN
```

Listing 4.1. Active program for a stateful firewall.

```
LOAD_TCP_FLAGS
CONST_LOAD_MBR2,$(2)
BIT_AND_MBR_MBR2
CJUMPI,@LOC_1
RETURN,:LOC_1
ADDR_OFFSET
TSLOAD_COARSE
COPY_MBR2_MBR
MEM_READ,#last_ts
SWAP_MBR_MBR2
MBR_SUBTRACT_MBR2
CONST_LOAD_MBR2,$(16000)
MIN
MBR_EQUALS_MBR2
CJUMP,@LOC_2
DROP,:LOC_2
MEM_WRITE,#last_ts
RETURN
```

Listing 4.2. Active program for a TCP SYN rate limiter.

timestamp into most-significant and least-significant 32-bits. The program simply checks if the timestamp of the new packet is within a second of the last recorded one, and drops the packet if it is. To prevent a burst of packets from going through (due to the architecture of RMT pipelines), we use an atomic primitive where we swap the last timestamp with a new one, every time we read the register.

Maglev Load Balancer

We attempted to implement a widely-used load balancer using vRMT. Such a load balancer is ideal for devices with constrained memory, due to its fixed-sized lookup tables. Maglev requires a table size to be a prime number. To allow such an application to be used in scenarios where virtual address translation is used, we require the table size to be a power of two. Hence, we choose an actual table size which is a Mersenne prime and omit one entry on the switch. Listing 4.3 shows the corresponding active program.

```
        LOAD_TCP_5TUPLE
        COPY_MBR2_MBR
        COPY_MAR_MBR2
        ADDR_MASK
        ADDR_OFFSET
        COPY_MBR2_MAR
        MEM_INCREMENT,#counter
        MEM_READ,#conn
        CJUMP,@LOC_1
        SET_IP_DST
        RETURN,:LOC_1
        COPY_MAR_MBR2
        ADDR_MASK
        ADDR_OFFSET
        MEM_READ,#backends
        SET_IP_DST
        COPY_MAR_MBR2
        MEM_WRITE,#conn
        RETURN
```

Listing 4.3. Active program for a Maglev load balancer.

Three stateful objects are used in this program – a `counter` to count the flow packets, a `conn` object to store connections and a `backends` table to store the backend server pool. An instance of this load balancer is associated with a single virtual IP (VIP); Additional instances can be provisioned for each VIP. Consistent with the original algorithm, the program first checks if the connection is present in the table and forwards it accordingly (using `SET_IP_DST`), else it consults the backend pool and installs a new entry. Additional *child* programs are created that perform tasks such as connection aging, expiry and backend pool update.

4.4 Recirculation-to-Completion

Packet processing in our model runs to completion by leveraging packet recirculations. Even when programs are shorter than the number of stages in a pipeline, function chaining requires every function to initiate in a new pass through the switch necessitating the use of packet recirculations. In this section, we first describe our approach to leveraging packet recirculations for function processing. We then evaluate whether packet recirculations on PISA pipelines can be harnessed as a technique to achieve run-to-completion for vRMT in a scalable and efficient manner coupled with end-to-end congestion control. To our knowledge, this is a first attempt to characterize the impact of packet recirculations on application performance over programmable switches.

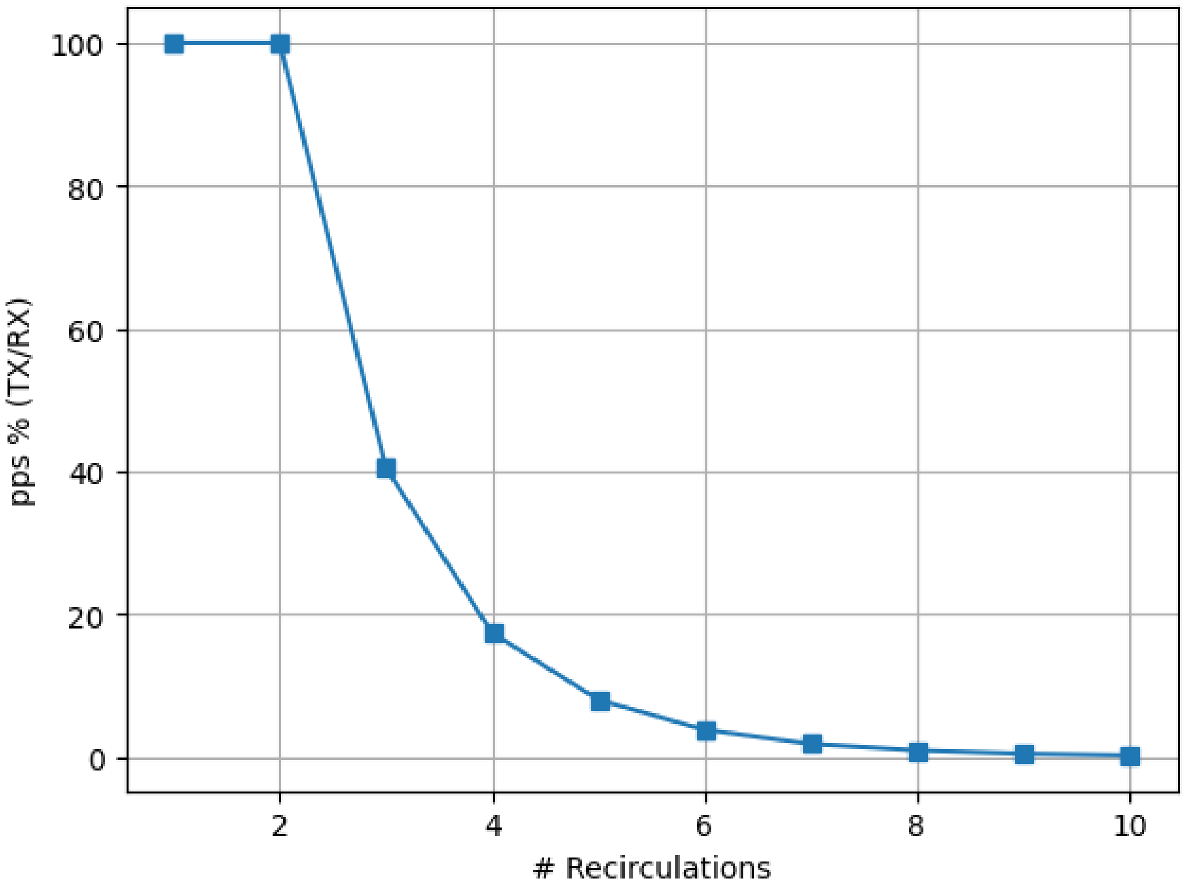


Figure 4.8. Throughput as a percentage of ingress traffic arriving at 100 Gbps, for varying number of recirculations over a 200 Gbps internal recirculation port on a Tofino switch.

On a Tofino device, each pipeline has 200 Gbps of recirculation bandwidth, which can efficiently recirculate packets over the switch backplane without requiring additional memory copying. However, such limited fixed bandwidth on a 1.6 Tbps pipeline does not allow us to scale in-network compute with packet recirculations. Inadequate bandwidth can lead to an exponential increase in overhead and ultimately a congestion collapse. Figure 4.8 illustrates the throughput achieved versus the number of times a packet is recirculated over Tofino's special recirculation port. In this experiment, we send traffic at 100 Gbps line rate on one of the physical ports. The switch is programmed to recirculate the traffic a certain number of times before transmitting it out on the same port. Since the device is provisioned with 200 Gbps of recirculation bandwidth for all traffic arriving at the special recirculation port, it can handle 100 Gbps of traffic with up to 2 recirculations. Soon after, the throughput degrades rapidly due to accumulating effects of congestion.

4.4.1 Virtualizing Bandwidth

We thus devise a method of leveraging (unused) device ports for packet recirculations, by configuring them to operate in loopback mode. A packet scheduled for recirculation is prepended with a metadata header to preserve state across recirculations, as packets leave the switch pipeline. The amount of metadata overhead used for packet recirculations is however limited by per-port capacities as dictated by the switching fabric. Exceeding this overhead results in packet drops. We use a 32B metadata header across recirculations.

To efficiently utilize the recirculation bandwidth, we combine all loopback ports into a link-aggregation-group and spray traffic across them. A special device port is configured for recirculations. At the end of ingress processing, packets that require recirculations are scheduled to this port and subsequently sprayed across all member ports – packet spraying is known to work well in general for balancing load. Packet recirculation is determined based on the function chain and program counter (which also determines the recirculation queue). Packets that have completed execution or are in their final pass are scheduled to an egress port determined by operator-specified forwarding rules. This also guarantees performance isolation among regular traffic and active traffic and consequently ensures availability of the network in the event of excessive backplane processing – recirculated traffic never congests regular traffic. Figure 4.9 illustrates this mechanism.

Packet recirculation has a variable overhead which depend on both packet-size and the functionality. When recirculating packets within the same device, we embed additional metadata containing program state[4]; per-packet state cannot be stored on the switch otherwise. This effectively inflates the bandwidth consumed by the original ingress traffic. This implies that for 100G traffic consisting of 64B packets arriving at line-rate on one of the switch ports, 150G of recirculation bandwidth has to be provisioned for each recirculation in the worst case. In practice, the amount of recirculated traffic is expected to be much lower – for throughput maximizing flows such as TCP this overhead would be 102G (considering a MTU size of 1500B). Recirculation overhead is thus heavily workload dependent.

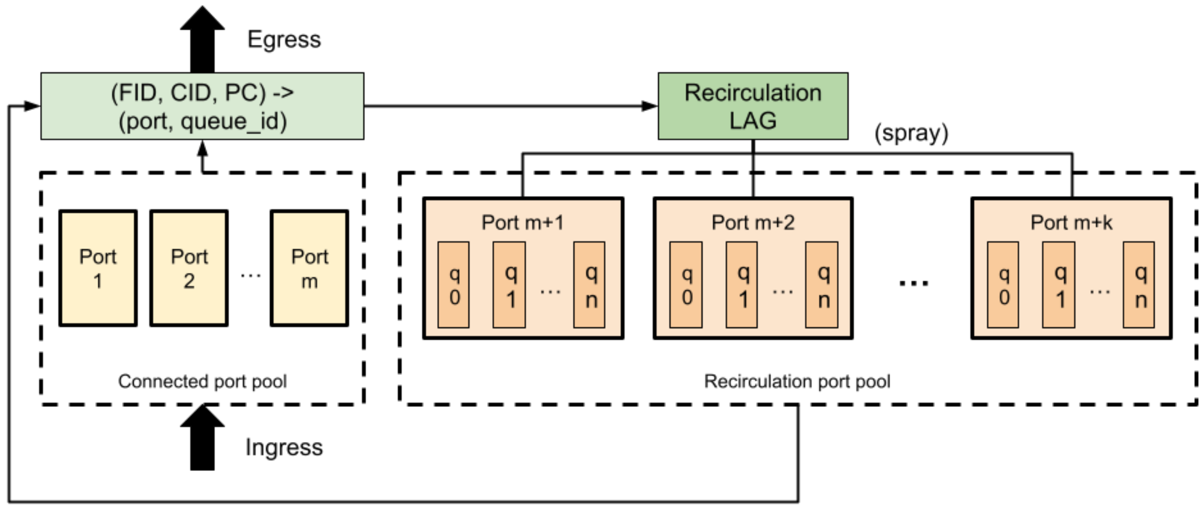


Figure 4.9. Packet scheduling mechanism for leveraging recirculation bandwidth to execute programs. Packets scheduled for recirculation are sprayed to a pool of recirculation ports.

Packet recirculations have been used extensively as an ability to run complex packet processing in prior work [8, 24, 27, 77]. Typically read-modify-write operations require additional recirculations [77]. Even while using static analysis to compose functions chains [27], the resulting programs grow in complexity and require recirculations. We now attempt to evaluate how such a feature can be made tractable on current programmable switches.

We first attempt to understand what impact backplane congestion has on application performance. We also attempt to answer what impact function chaining has on application per-

[4]Most switches allow creation of new packet headers.

formance and fairness. Table 4.1 lists a set of functions that we implemented using vRMT along with the minimum and maximum number of recirculations that are required to execute each of them. The number of recirculations is application-dependent. For example, in the load balancer, only new connections require 2 recirculations; All other packets require one recirculation.

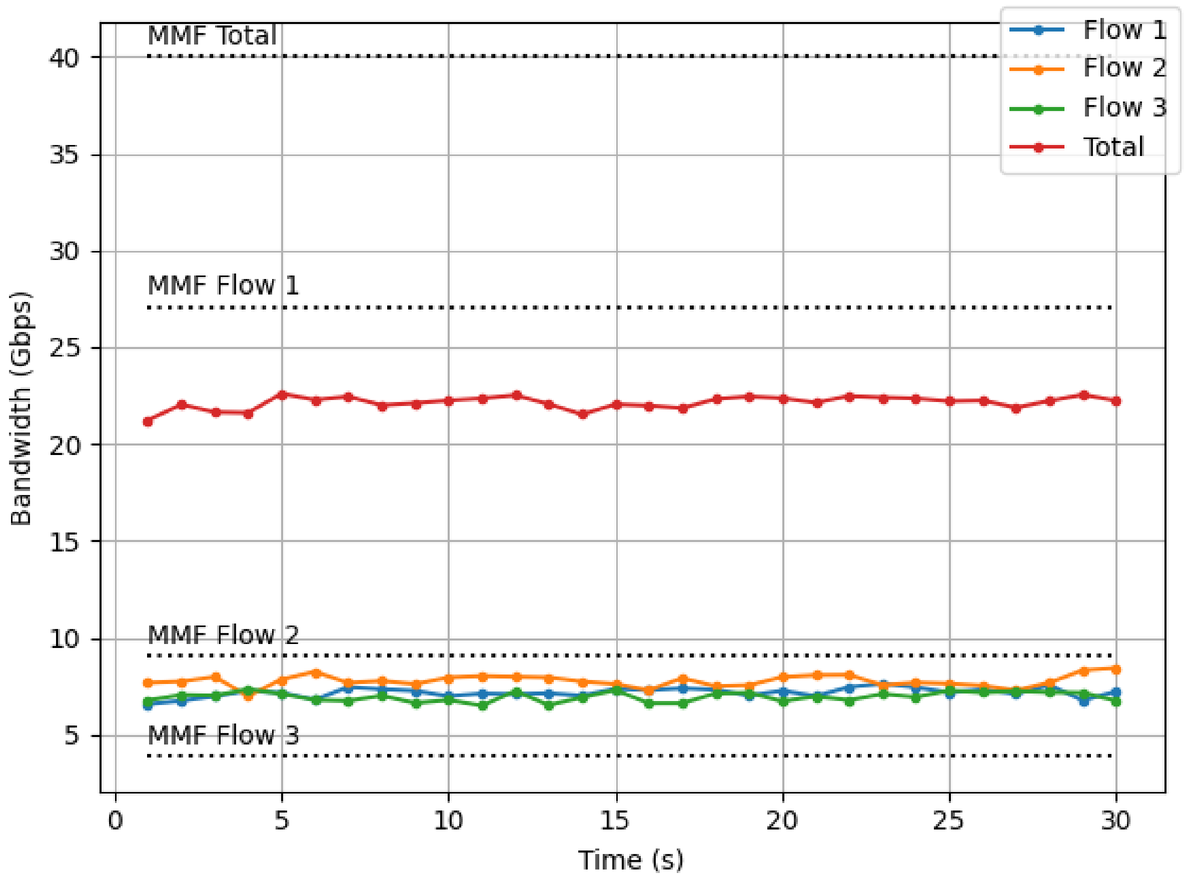


Figure 4.10. Throughput and stability of three flows with 1,3,7 recirculations respectively on a 40G link, sharing a shared droptail queue on a recirculation port with 100 Gbps capacity. Flows show instability with a maximum variation of 1 Gbps. Total throughput remains well below the link rate.

4.4.2 Loopback Congestion

Queueing plays an significant role in fairness and efficiency in such a network. We assign a distinct recirculation queue to each function chain; Separation of queues isolate the effects of congestion across flows and allow flows to achieve a proportional share of the recirculation bandwidth. An alternative method of assigning each function a distinct queue could lead to applications unfairly obtaining a larger share of their bandwidth.

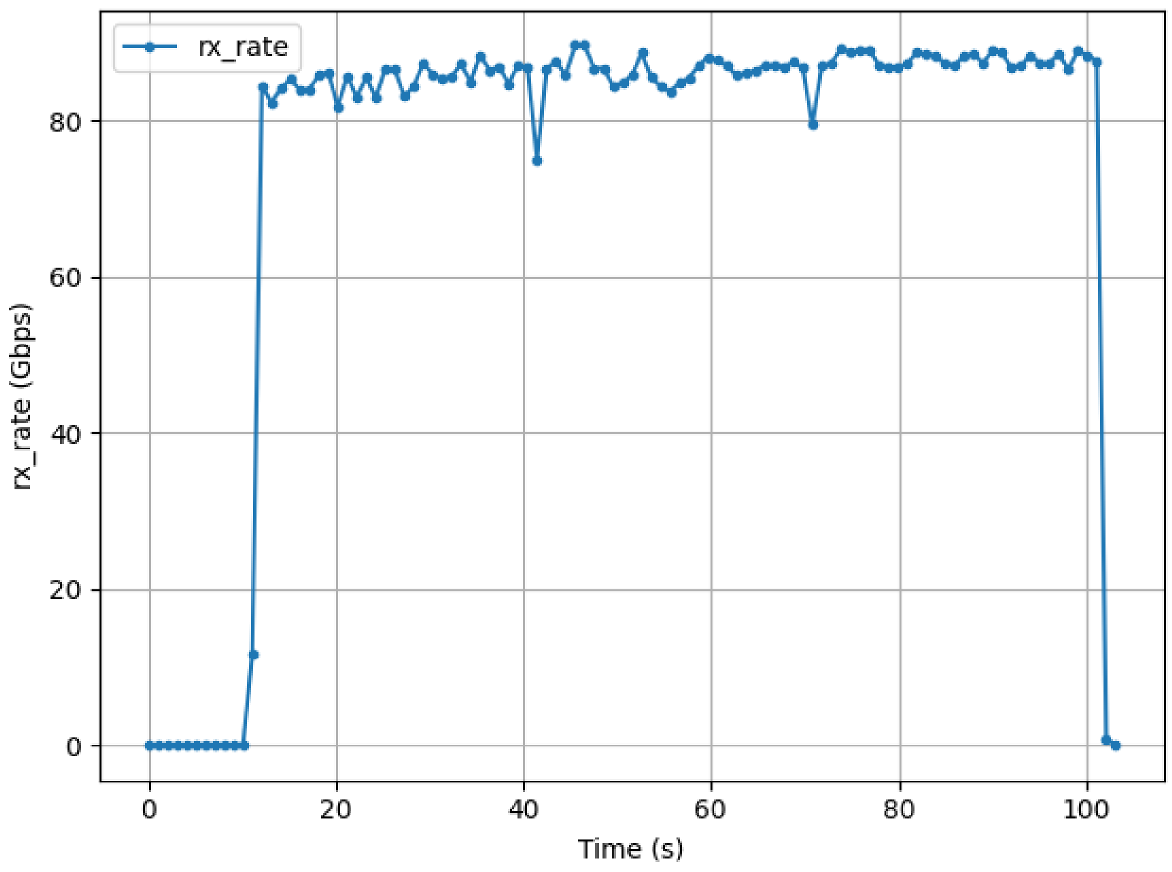


Figure 4.11. Recirculation bandwidth usage in the presence of droptail queues. Three flows with 1,3,7 recirculations respectively, share a shared droptail queue on a recirculation port with 100 Gbps capacity. TCP cubic flows ingress the switch on a 40 Gbps link.

Unlike regular network traffic, recirculated packets are enqueued multiple times on the same queue. The propagation delay for each recirculation is less than the one-way-delay of packets through the network, resulting in sudden increase in queue occupancy at times of congestion. While this is equivalent to experiencing congestion from cross-traffic, the sources in this case are not independent and will not throttle their rates independently in a decentralized manner – cross-traffic is induced from packet recirculations. Consequently, the loss probability is not deterministic and depends on the current queue occupancy (which subsequently depends on the number of recirculations). When using shared drop-tail queues this causes flows to converge to an equal share of the link bandwidth and lead to instability. Figure 4.10 shows the bandwidth allocated to three flows (each corresponding to a distinct application) with 1, 3 and 7 recirculations respectively on a 40G link with shared queues using TCP cubic. The aggregate recirculation bandwidth required (11 x 13 = 143 Gbps) for a max-min fair allocation on the ingress link exceeds the capacity of the recirculation port. TCP adjusts the send rates such that each get a much lower share of bandwidth that stays around 7.5 Gbps (which is close to the max-min fair allocation of 7.33 Gbps each). The recirculation bandwidth is also underutilized and unstable as shown in Figure 4.11.

Not only is the link significantly underutilized in such a setting, attempting to give each flow an equal share of the link bandwidth leads to flows with fewer number of recirculations getting affected by flows with higher number of recirculations, in the presence of congestion. This is undesirable since flows with high number of recirculations could severely degrade the network utility. A fair (and strategy-proof) allocation in such a scenario would allocate bandwidth to be inversely proportional to the number of recirculations – a weighted max-min fair allocation. For example, in the above scenario, traffic would be split as 14.9 Gbps, 5 Gbps and 2.1 Gbps respectively using a weighted max-min-fair allocation. Such an allocation scheme naturally forces application services to make a compromise between network usage and functional utility – more complex applications would get a lesser share of the line bandwidth in events of congestion; Recirculation bandwidth is evenly split among applications, effectively giving them equal processing resources. Using dedicated queues for each application (we expect the number of concurrent applications to be small) isolates the congestion effects of applications from each other and leads to higher overall throughput and more stable throughput per application. Figure 4.12 compares the throughputs observed using droptail and RED queueing schemes. In each experiment, a set of 5 concurrent flows were started consuming 1-5 recirculations respectively. The plots show the expected weighted max-min-fair throughputs on a 40G link. Using a configuration of RED tuned, we are able to give flows with lower number of recirculations a higher share of bandwidth and close to their fair share. Recirculation bandwidth is also utilized efficiently and congestion isolated among the queues.

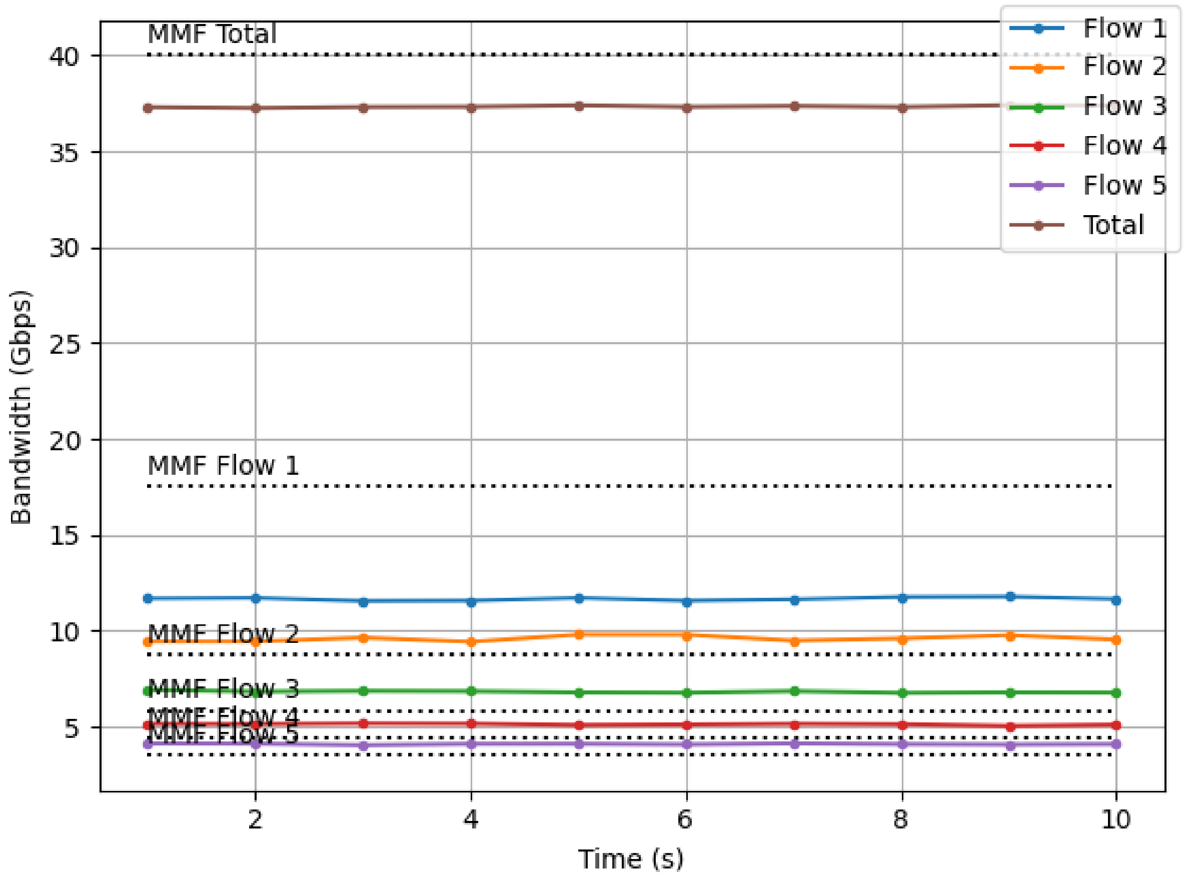


Figure 4.12. Throughputs of 5 flows with dedicated droptail queues on a 40G link. A 100G recirculation bandwidth port was used in this experiment. Flows with fewer number of recirculations get a significantly less share of their fair share of link bandwidth.

4.4.3 Buffer Contention

We would want to reduce the effect of cross-traffic congestion and its impact on the fairness of application flows, especially those that use loss-based congestion control. Unfortunately, switches such as the Tofino have shallow buffers ($\approx$20 MB). Ports are assigned a minimum buffer capacity along with a shared headroom for bursts. With recirculations rapidly filling up queues, this results in a race condition. Limiting the maximum buffer usage for a flow could potentially alleviate this problem. A known way to achieve this is to use AQM. Figure 4.12 shows the throughputs for regular dedicated queues and RED (Figure 4.13), for the same configuration. A 0.001% marking probability is used with a packets marked according to the RED scheme from 100 to 1000 cells, where each cell is 80B. A 1-μS window is used to compute average queue size. RED is able to give a fairer share of bandwidth to flows with fewer recirculations.

However, queues are pinned to application functions or chains instead of flows. Hence, there would still be contention among multiple flows sharing the same queue. However, since they are all expected to have the same number of recirculations, the loss probabilities are similar and they would then get an equal share of bandwidth.

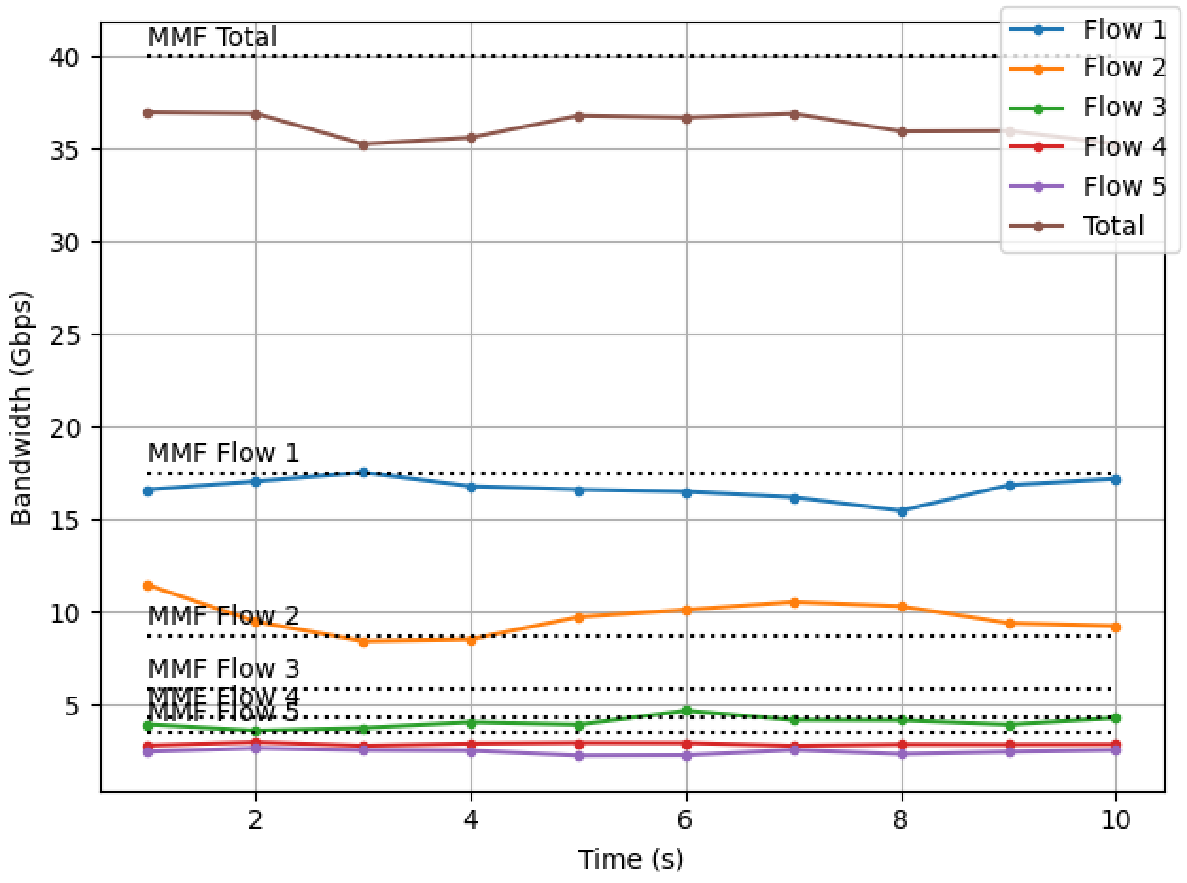
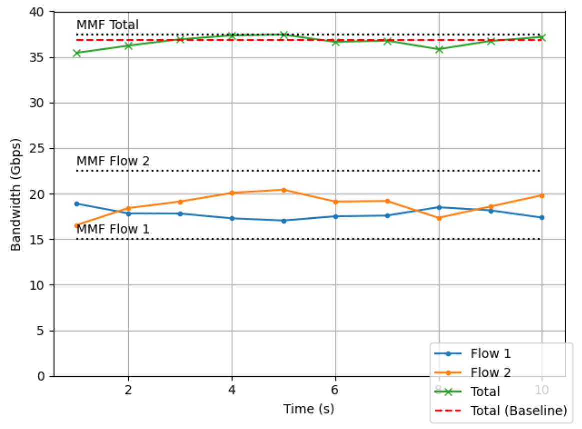
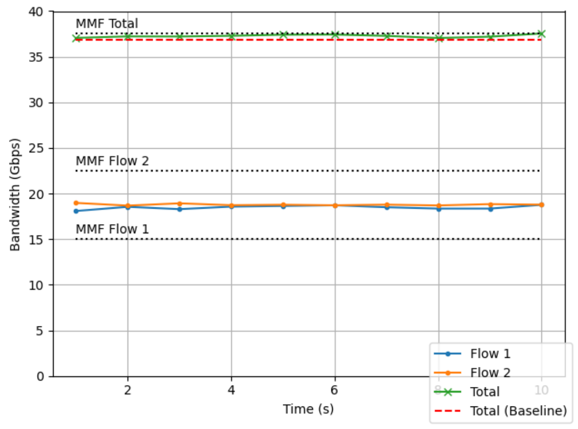


Figure 4.13. Throughputs of 5 flows with tuned dedicated RED queues on a 40G link. A 100G recirculation bandwidth port was used in this experiment. Flows with fewer number of recirculations get a higher share of the link bandwidth (close to their fair share).



(a) 100G recirculation BW.



(b) 200G recirculation BW.

Figure 4.14. Impact of recirculation bandwidth on fairness. iPerf performance of two concurrent flows on a 40G link.

4.4.4 Provisioning Bandwidth

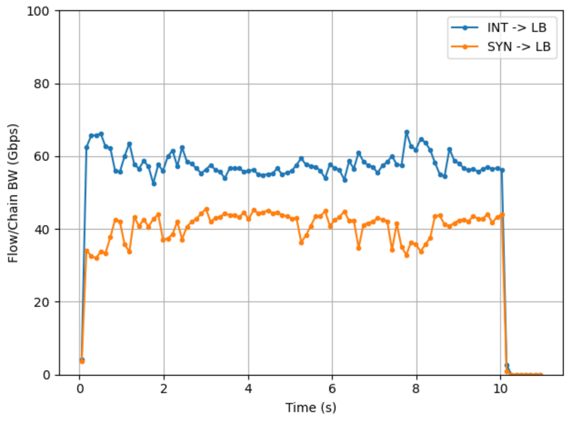
We devise a system that is able to re-purpose connected device ports for packet recirculations, effectively increasing the processing capacity of the switch. While one can statically provision bandwidth along with the application, we note that bandwidth consumption cannot be exactly determined ahead of time. Function chains are executed in the order in which they are stored and consume the sum of the number of recirculations in bandwidth in the worst case. However, in practice functions may terminate early and the number of recirculations consumed by the chain would be fewer. Similarly, functions that drop packets (e.g. an ACL) would terminate chains earlier. We now take a look at the impact and sensitivity of additional recirculation headroom on the network performance of application flows, in order to determine the potential for dynamic provisioning of such bandwidth. We evaluate network performance in scenarios where server applications communicate over a programmable network running in-network functions.

We use a mechanism where function chains are pinned to distinct queues. A flow is typically assigned to a single function chain and hence all packets of the same flow experience the same performance. Congestion can have significant effects on fairness and throughput of competing application flows. To this end we performed an experiment using three in-network applications – in-network telemetry (INT), syn rate limiter (SYN) and a Maglev load balancer (LB). INT and SYN consume 2 iterations each in the worst case, while LB consumes 3; The SYN and LB functions consume 1 and 2 iterations for non SYN packets. Packet filters for these functions were chosen in such a way that they form two distinct chains with INT and SYN belonging to a separate chain and the LB sharing both chains. Hence, the chains consume 3 and 2 recirculations respectively (the last iteration in each chain is not recirculated). Flow 1 is chained to INT and LB with 3 recirculations while flow 2 is chained to SYN and LB with 2 recirculations. With enough recirculation bandwidth, flows can get an equal share of the link; Otherwise their shares are split inversely proportional to the number of recirculations they consume. The baseline shows the maximum link throughput obtained with two flows in the absence of any in-network functions. Figure 4.14 shows the performance of the iPerf flows. This phenomenon is made more evident by observing the recirculation bandwidth usage by chain as shown in Figure 4.15. Ideally flows are meant to get an equal share of the recirculation bandwidth during times of congestion, although buffer contention leads to otherwise (Figure 4.16). Increasing the recirculation capacity to 200G increased the recirculation bandwidth usage from 99.8 Gbps to 101.6 Gbps – an $\approx$2% increase in recirculation bandwidth can improve application performance (and increase the fair share of an affected flow by $\approx$10%). This sets the case for dynamic allocation of recirculation bandwidth (even a small increase in bandwidth can improve fairness).

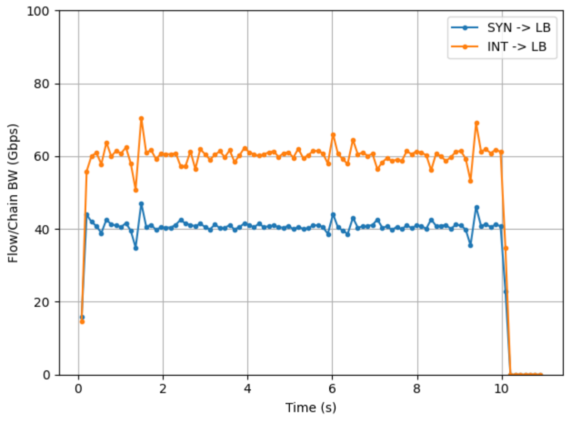
4.4.5 Application Benchmarks

While our previous evaluations have focused on consistent high-throughput scenarios, in practice, real application flows affected by in-network computation may exhibit other patterns – e.g. short-lived flows. We aim to determine how much impact the effects of recirculation-congestion has on real application performance. To this end, we run performance benchmarks on two real TCP applications – a HTTP server and an in-memory database.

We evaluate the impact of packet recirculations on server applications that use the

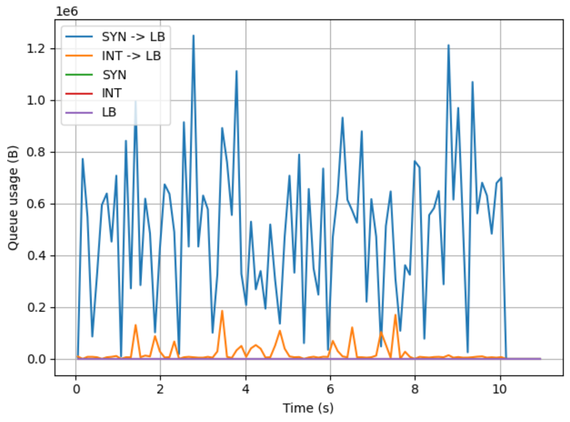


(a) 100G recirculation BW.

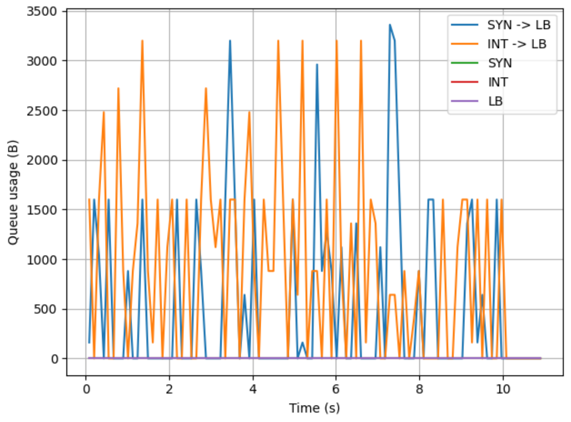


(b) 200G recirculation BW.

Figure 4.15. Recirculation bandwidth usage by function chain.

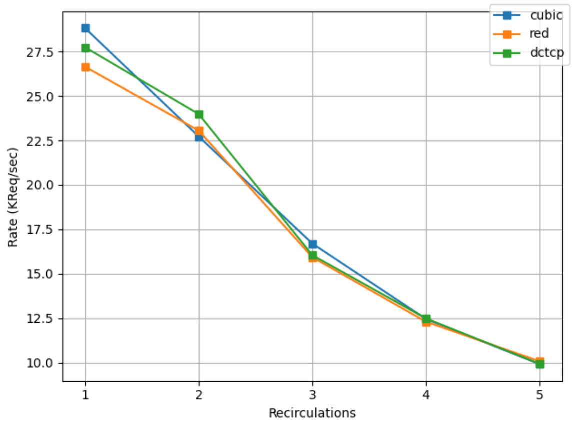


(a) 100G recirculation BW.

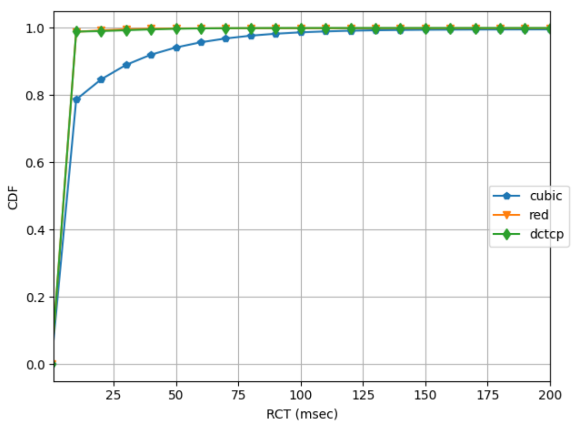


(b) 200G recirculation BW.

Figure 4.16. Queue utilization on loopback ports of function chains pinned to distinct recirculation queues.



(a) WRK HTTP benchmark.



(b) Redis reads.

Figure 4.17. Application benchmarks using DCTCP, Cubic and RED (w/ cubic) over a vRMT network.

vRMT network, using various congestion control algorithms. We use WRK [32] to evaluate HTTP performance. We run multiple concurrent instances of the tool for each in-network application requiring 1-5 recirculations respectively on the downlink. A total of 400 concurrent connections and 2 threads were used for each application. A fixed sized web page was served by the server for each request. The experiment runs over 10 seconds and saturates a 40G downlink to the clients. Figure 4.17a shows the throughputs for each cluster of flows corresponding to the respective application in requests per second against varying number of recirculations.

We also evaluate a key-value database performance using Redis. We use 5 servers, each hosting an independent set of objects with sizes following a known datacenter cache distribution [73]. We use 5 corresponding clients using 64 threads each that concurrently make random requests to the respective servers using keys drawn from the same distribution. This forces the flow sizes to follow the corresponding distribution. As before, each of these instances incur 1-5 recirculations respectively on a 40G downlink and the responses saturate the link. We measured the time to complete a request (which includes the end-host overheads) – we refer to this as the request completion time (RCT). Figure 4.17b shows the RCTs for each of the congestion control mechanisms used in the experiment. The parameters for DCTCP and RED are the same as before. We see that ECN/RED reduce the RCTs significantly as compared to Cubic which is widely used across hosts. A total of 100G recirculation bandwidth was provisioned in each case.

4.5 Discussion

We present a software defined network that allows network functions that perform behavioral inspection to co-exist with application offloads. An online mechanism composes functions from arbitrary sources into function chains. A corresponding online packet classification scheme that runs at line rate can hold hundreds of typical packet filter rules, and is used to assign packets to function chains. With a few tweaks to queuing, we are able to utilize switch backplane capacity to make recirculation-to-completion tractable and help achieve efficient utilization and fairness among competing application flows using end-to-end congestion control. Our results indicate that with our authenticated classification scheme, we can potentially accommodate hundreds of Snort-like rules corresponding to application functions, over a runtime that supports 12 execution stages.

Acknowledgements

Chapter 4 contains material that is a part of a manuscript being prepared for submission. The dissertation author was the primary investigator and author of this manuscript.

Chapter 5
Conclusion

We show that programmable RMT pipelines can be virtualized to support multiprocessing, runtime programmability and dynamic resource allocation. While a capsule-based approach to active networking can be used to achieve fine-grained multiprocessing, a more robust solution in the likes of virtual network functions could enable a broader set of functionality to be deployed over such a programmable network. Leveraging P4 and RMT as platforms, we thus present two systems – ActiveRMT and vRMT – that export high-level programmability to application programmers on virtualized modern programmable switch hardware based on RMT. Our prototype implementations of ActiveRMT and vRMT are built to run on a Tofino programmable switch. We demonstrate that dynamic scheduling of instructions to virtualized processors can make efficient use of switch memory. In addition, function chain execution over virtualized switches can be achieved using domain switching among provisioned network functions. An online classifier can be used to facilitate such a capability, which can be realized with a minimal footprint on RMT hardware. In addition, backplane bandwidth can be virtualized to perform packet processing – a technique we refer to as recirculation-to-completion – which can be used to improve application performance dynamically. Our systems however, have several limitations, primarily with respect to the underlying hardware. They also open up avenues for future work. We discuss them here.

5.1 Limitations

RMT hardware enables high-performance (and high-capacity) devices to be reconfigured at timescales typically in the order of minutes, much faster than any alternative. The hardware on such devices are designed to enable general purpose utilization based on common network functional characteristics. However, when provisioning such hardware for a general-purpose execution environment, resource utilization is suboptimal, subsequently resulting in suboptimal capabilities.

For instance, PHV containers provide context for a packet during the lifetime of the packet within the switch; Internal program state corresponding to active programs is stored within PHV containers. Achieving a connected virtual packet processing pipeline however, requires some of that state to be shared across all virtual instruction processing tables (for example, the `MAR` register variable has to be shared across all virtual processor stages). However, due to the limited size of those container groups (and certain other device constraints), the total size of the shared internal state is also limited – this includes program variables such as `MAR` and also extracted program arguments. This subsequently, results in a tradeoff between the size of memory words and the maximum number of state variables (including program arguments) that can be supported – our implementation uses 32-bit memory words for instance, which could be replaced with 16-bit words and more program arguments. While some workarounds to such a limitation are indeed possible – such as the use of multiplexing stages in vRMT, they are also associated with tradeoffs. A multiplexing stage for example, reduces functional redundancy by taking away from a virtual processing stage. It also reduces resource efficiency (e.g. memory) and subsequently increases the total number of stages required to execute an active program.

Another notable concern is associated with the efficient packing of functionality to switch resources. The pre-configuration of switch pipelines to perform general purpose computation has unavoidable overheads. While we do better than prior approaches to virtualization on RMT, there may be possibilities to do even better in terms of overheads. 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.

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 statically. For example, in ActiveRMT, 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. Alternatively, switch ASICs could implement support for such standardized protocols within fixed functions, while exporting RMT hardware for custom functionality. Our systems provide a basis for expressing arbitrary forwarding functionality, such as source routing. Overriding operator-defined forwarding behavior may however, be a security concern. Our systems however, enable protection domains that allow only trusted users to execute such functionality.

5.2 Alternative Hardware

Memory in RMT is stage-local – SRAM in a match-action stage cannot be accessed by any other stage. Such a design results in less efficient use of memory in a number of cases – for example, when there are dependencies. Alternative architectures such as dRMT alleviates this problem by decoupling memory from the match-action stages and assigns them to the stages at compile time. However, only NICs have currently adopted this variant and their feasibility on programmable switches is yet unknown. Such a memory model can however improve memory efficiency for both ActiveRMT and vRMT. Due to the overheads of preprocessing (active) packets including classification, chaining and authentication a number of RMT stages have unutilized memory. Decoupling memory from processing stages can allow assigning more device memory to the virtual processing stages.

Both our systems demonstrate that using pre-configured match tables can be used to express complex programs. Such a design advocates for the use of pre-defined multiple-match-tables (MMT). A runtime can be configured using such hardware by choosing the set of instructions that are enabled at each stage. However, the benefits of RMT still remain. As an example, classifiers can be configured with an arbitrary number of bits for classes based on the domain of rules that will be installed on such devices. RMT also allows operators to specify the precision of computation (e.g. 16-bit words as compared to 32-bit words). Current device externs such as register ALUs and hash units can be customized to match the runtime. A set of features that enable operator customization of the runtime should be separate from ones that enable user programmability.

While run-to-completion processors are another viable processing architecture, we believe their true utility lies in the ability to perform complex read-modify-writes. General purpose processors have limited utility in such a design since most networks are meant to be best-effort. Any processing architecture must take into consideration the reliability of the network when considering programmability – there is no point performing highly complex computations in the network only to drop packets later on due to congestion; At the very least such processors should only be assigned to egress pipelines, post queueing.

5.3 Network-Wide Planning

We currently consider deployment of network functions on a single switch. Such an approach can however be generalized to an entire software-defined network. This requires us to consider all virtual resources, including total switch backplane processing bandwidth and virtualized memory. Provisioning based on the former may be more tractable – this requires us to include additional constraints in an existing network planner (typically involving a max-min fair solver); Using a cost function that is linear is necessary to achieve such a goal. We would ideally want to optimize for both link bandwidth and virtualized recirculation bandwidth to make most efficient use of switch backplane capacity. This requires us to estimate the volume of recirculated traffic on each switch, the connected forwarding capacity of each switch and additional recirculation capacity.

Considering virtualized memory however, makes this problem much harder, since such resources are not fluid and cannot be linearized due to program dependency constraints. Using our current model, resources that affect such a planning decision include table memory, register memory, recirculation bandwidth and the number of queues. Hence, a potential plan is one that selects a switch from a set of potential switches which maximizes packing, minimizes congestion, achieves minimum variance among number of recirculations in flows assigned to a queue. Non-linear switch resources (e.g. register memory) can be addressed by augmenting a trial-and-error (or exploration-exploitation) approach. Such plans may be solved globally using a weighted max-min-fair allocator.

5.4 Language Extensions

Our instruction set is expressive enough to implement a number of useful functions. However, a higher level language may be desirable. To this end we have made some initial developments towards an imperative-style language that supports semantics with data types to generalize common functional capabilities on switches. For instance, register memory is accessible via a `mem_t` data type. Assignments to and from such data variables implement register memory access functionality. Another such data type is the `hash_t` type which implements hashing. Such types can be extended to support data structures such as sets, dictionaries and buckets using the underlying vRMT architecture.

Our current systems can be used to perform operations such as remote memory access on switch memory and object storage. The language system can also be augmented with the control plane to enable additional capabilities faciliated by current devices such as Tofino. This includes event generation (e.g. in order to signal congestion from the switch), rate limiters, scheduling, timing (e.g. to implement a periodic health checker). Such capabilities can be used to leverage richer application functionality with minimal effort from application programmers.

Alternatively, ActiveRMT and vRMT may be integrated into P4. However, certain constructs such as match-action tables cannot be efficiently expressed using such a virtualized runtime. Yet, one may consider implementing ActiveRMT or vRMT as an “extern” – a feature used to export hardware-specific capabilities to programmers – on P4 target architectures. Such an extern may be programmed independently using language semantics borrowed from P4. We leave all such explorations to future work.

Bibliography

[1] Anurag Agrawal and Changhoon Kim. “Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch”. In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. Palo Alto, CA, USA, August 2020, pp. 1–32.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Workload Analysis of a Large-Scale Key-Value Store”. In: *ACM SIGMETRICS Performance Evaluation Review* 40.1 (June 2012), pp. 53–64.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. “Fast Userspace Packet Processing”. In: *Proceedings of the 11th ACM/IEEE Symposium on Architectures for networking and communications systems*. Oakland, CA, USA, May 2015, pp. 5–16.

[4] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. “A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency”. In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, USA, February 2020, pp. 667–683.

[5] Barefoot Networks. *barefootnetworks/Open-Tofino*. https://github.com/barefootnetworks/Open-Tofino.

[6] Antonin Bas. *Leveraging Stratum and Tofino Fast Refresh for Software Upgrades*. https://opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf. December 2018.

[7] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. "Designing Heavy-Hitter Detection Algorithms for Programmable Switches". In: *IEEE/ACM Transactions on Networking* 28.3 (June 2020), pp. 1172–1185.

[8] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. "PINT: Probabilistic In-band Network Telemetry". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Virtual Event, USA, July 2020, pp. 662–680.

[9] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. "Active Networking and the End-to-End Argument". In: *Proceedings of the International Conference on Network Protocols*. Atlanta, GA, USA, October 1997, pp. 220–228.

[10] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch". In: *ACM SIGCOMM Computer Communication Review* 44.2 (April 2014), pp. 44–51.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-independent Packet Processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95.

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Hong Kong, China, August 2013, pp. 99–110.

[13] Anat Bremler-Barr, Yotam Harchol, and David Hay. "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Florianópolis, Brazil, August 2016, pp. 511–524.

[14] Max A. Cherney. *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. January 2023.

[15] 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. "dRMT: Disaggregated Programmable Switching". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Los Angeles, CA, USA, August 2017, pp. 1–14.

[16] Cisco Systems. *Snort - Network Intrusion Detection & Prevention System*. https://www.snort.org/.

[17] Graham Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and its Applications". In: *Journal of Algorithms* 55.1 (April 2005), pp. 58–75.

[18] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. "Paxos Made Switch-y". In: *ACM SIGCOMM Computer Communication Review* 46.2 (May 2016), pp. 18–24.

[19] Emilie Danna, Subhasree Mandal, and Arjun Singh. "A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering". In: *Proceedings of IEEE INFOCOM*. Orlando, FL, USA, March 2012, pp. 846–854.

[20] Kevin Deierling. "NVIDIA's Resource Transmutable Network Processing ASIC". In: *IEEE Hot Chips 35 Symposium (HCS)*. Palo Alto, CA, USA, August 2023, pp. 1–14.

[21] DPDK Project. *DPDK*. https://www.dpdk.org/about/.

[22] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *Proceedings*

of the 13th USENIX Symposium on Networked Systems Design and Implementation. Santa Clara, CA, USA, March 2016, pp. 523–535.

[23] Nick Feamster, Jennifer Rexford, and Ellen Zegura. “The Road to SDN: An Intellectual History of Programmable Networks”. In: *ACM SIGCOMM Computer Communication Review* 44.2 (April 2014), pp. 87–98.

[24] Yong Feng, Zhikang Chen, Haoyu Song, Yinchao Zhang, Hanyi Zhou, Ruoyu Sun, Wenkuo Dong, Peng Lu, Shuxin Liu, Chuwen Zhang, Yang Xu, and Bin Liu. “Empower Programmable Pipeline for Advanced Stateful Packet Processing”. In: *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, USA, April 2024, pp. 491–508.

[25] Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. “In-situ Programmable Switching using rP4: Towards Runtime Data Plane Programmability”. In: *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. Virtual Event, USA, November 2021, pp. 69–76.

[26] Jonas Fietz, Sam Whitlock, George Ioannidis, Katerina Argyraki, and Edouard Bugnion. “VNToR: Network Virtualization at the Top-of-Rack Switch”. In: *Proceedings of the 7th ACM Symposium on Cloud Computing*. Santa Clara, CA, USA, October 2016, pp. 428–441.

[27] Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai. “Sirius: Composing Network Function Chains into P4-Capable Edge Gateways”. In: *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, USA, April 2024, pp. 477–490.

[28] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. “Switch Code Generation Using Program Synthesis”. In: *Proceedings of the*

Conference of the ACM Special Interest Group on Data Communication. Virtual Event, USA, July 2020, pp. 44–61.

[29] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. "Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds". In: *arXiv:1305.0209 [cs]* (March 2014).

[30] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. "OpenNF: Enabling Innovation in Network Function Control". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Vol. 44. Chicago, IL, USA, August 2014, pp. 163–174.

[31] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. Boston, MA, USA, March 2011, pp. 323–336.

[32] Will Glozer. *wg/wrk*. https://github.com/wg/wrk.

[33] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. "Sonata: Query-driven Streaming Network Telemetry". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Budapest, Hungary, August 2018, pp. 357–371.

[34] Pankaj Gupta and Nick McKeown. "Packet Classification on Multiple Fields". In: *ACM SIGCOMM Computer Communication Review* 29.4 (August 1999), pp. 147–160.

[35] Pankaj Gupta and Nick McKeown. "Algorithms for Packet Classification". In: *IEEE Network* 15.2 (April 2001), pp. 24–32.

[36] David Hancock and Jacobus van der Merwe. “HyPer4: Using P4 to Virtualize the Programmable Data Plane”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. Irvine, CA, USA, December 2016, pp. 35–49.

[37] Linda Hardesty. *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/. August 2018.

[38] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. “A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research”. In: *arXiv:2101.10632 [cs.NI]* (August 2021).

[39] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. “Modular Switch Programming Under Resource Constraints”. In: *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2022, pp. 193–207.

[40] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. “Elastic Switch Programming with P4All”. In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. Chicago, IL, USA, November 2020, pp. 168–174.

[41] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. “NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms”. In: *Proceeedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. Seattle, WA, USA, April 2014, pp. 445–458.

[42] Ofer Iny. *Cisco Silicon One Powers the Next-Generation Enterprise Switches*. blog. https://blogs.cisco.com/sp/cisco-silicon-one-powers-the-next-generation-enterprise-switches. February 2022.

[43] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems". In: *arXiv:cs/9809099 [cs.NI]* (September 1998).

[44] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Chicago, IL, USA, August 2014, pp. 3–14.

[45] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination". In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2018, pp. 35–49.

[46] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. Shanghai, China, October 2017, pp. 121–136.

[47] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. "Compiling Packet Programs to Reconfigurable Switches". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*. Oakland, CA, USA, May 2015, pp. 103–115.

[48] Georgios P. Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q. Maguire Jr. "Metron: NFV Service Chains at the True Speed of the Underlying Hardware". In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2018, pp. 171–186.

[49] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr, and Dejan Kostić. "SNF: synthesizing high performance NFV service chains". In: *PeerJ Computer Science* 2 (November 2016), e98.

[50] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. "Rate control for communication networks: shadow prices, proportional fairness and stability". In: *Journal of the Operational Research Society* 49.3 (March 1998), pp. 237–252.

[51] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends". In: *IEEE Access* 9 (June 2021), pp. 87094–87155.

[52] Changhoon Kim, Anirudh Sivaraman, Naga Praveen Katta, Antonin Bas, Advait Dixit, and Lawrence J. Wobker. "In-band Network Telemetry via Programmable Dataplanes". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (Demos)*. London, United Kingdom, August 2015.

[53] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. "ExoPlane: An Operating System for On-Rack Switch Resource Augmentation". In: *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*. Boston, MA, USA, April 2023, pp. 1257–1272.

[54] Young-Deok Kim, Hyun-Seok Ahn, Suhwan Kim, and Deog-Kyoon Jeong. "A High-Speed Range-Matching TCAM for Storage-Efficient Packet Classification". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.6 (June 2009), pp. 1221–1230.

[55] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains". In: *IEEE/ACM Transactions on Networking* 28.2 (April 2020), pp. 639–652.

[56] T. V. Lakshman and D. Stiliadis. "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Vancouver, BC, Canada, October 1998, pp. 203–214.

[57] Alberto Lerner, Rana Hussein, and Philippe Cudre-Mauroux. "The Case for Network-Accelerated Query Processing". In: *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*. Monterey, CA, USA, January 2019, p. 10.

[58] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. "LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed". In: *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*. Boston, MA, USA, April 2023, pp. 1451–1468.

[59] Jialin Li, Ellis Michael, and Dan R. K. Ports. "Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. Shanghai, China, October 2017, pp. 104–120.

[60] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. "Be Fast, Cheap and in Control with SwitchKV". In: *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, USA, March 2016, pp. 31–44.

[61] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. "Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Budapest, Hungary, August 2018, pp. 504–517.

[62] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the Art of Network Function Virtualization". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. Seattle, WA, USA, April 2014, pp. 459–473.

[63] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In:

Proceedings of the Conference of the ACM Special Interest Group on Data Communication. Los Angeles, CA, USA, August 2017, pp. 15–28.

[64] Jeonghoon Mo and Jean Walrand. "Fair End-to-End Window-Based Congestion Control". In: *IEEE/ACM Transactions on Networking* 8.5 (October 2000), pp. 556–567.

[65] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. "Composing Software-Defined Networks". In: *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. Lombard, IL, USA, April 2013, pp. 1–14.

[66] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. "The Click modular router". In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Vol. 33. Charleston, SC, USA, December 1999, pp. 217–231.

[67] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. "SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches". In: *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2022, pp. 743–759.

[68] P4 Language Consortium. *P4-16 Language Specification*. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. 2017.

[69] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. "E2: A Framework for NFV Applications". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. Monterey, CA, USA, October 2015, pp. 121–136.

[70] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. Savannah, GA, USA, November 2016, pp. 203–216.

[71] Michał Pioro, Peter Nilsson, Eligijus Kubilinskas, and Gábor Fodor. “On Efficient Max-Min Fair Routing Algorithms”. In: *Proceedings of the 8th IEEE Symposium on Computers and Communications*. Kemer - Antalya, Turkey, July 2003, 365–372 vol.1.

[72] Bozidar Radunovic and Jean-Yves Le Boudec. “A Unified Framework for Max-Min and Min-Max Fairness With Applications”. In: *IEEE/ACM Transactions on Networking* 15.5 (October 2007), pp. 1073–1083.

[73] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. “Inside the Social Network’s (Datacenter) Network”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. London, United Kingdom, August 2015, pp. 123–137.

[74] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. “Scaling Distributed Machine Learning with In-Network Aggregation”. In: *arXiv:1903.06701 [cs, stat]* (February 2019).

[75] Junxian Shen, Heng Yu, Zhilong Zheng, Chen Sun, Mingwei Xu, and Jilong Wang. “Serpens: A High-Performance Serverless Platform for NFV”. In: *IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. Virtual Event, USA, June 2020, pp. 1–10.

[76] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Florianópolis, Brazil, August 2016, pp. 15–28.

[77] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. “Heavy-Hitter Detection Entirely in the Data Plane”. In: *Proceedings of the Symposium on SDN Research*. Santa Clara, CA, USA, April 2017, pp. 164–176.

[78] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. “Network Load Balancing with In-network Reordering Support for RDMA”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA, September 2023, pp. 816–831.

[79] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. “Composing Dataplane Programs with μP4”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Virtual Event, USA, July 2020, pp. 329–343.

[80] Vegesna S. M. Srinivasavarma, Shanmukha Rao Pydi, and S. Noor Mahammad. “Hardware-based multi-match packet classification in NIDS: an overview and novel extensions for improving the energy efficiency of TCAM-based classifiers”. In: *The Journal of Supercomputing* 78.11 (July 2022), pp. 13086–13121.

[81] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. *New Streaming Algorithms for Fast Detection of Superspreaders*. Tech. rep. Fort Belvoir, VA: Defense Technical Information Center, May 2004.

[82] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. “Quadrant: A Cloud-Deployable NF Virtualization Platform”. In: *Proceedings of the 13th Symposium on Cloud Computing*. San Francisco, CA, USA, November 2022, pp. 493–509.

[83] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. “Isolation Mechanisms for High-Speed Packet-Processing Pipelines”. In: *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2022, pp. 1289–1305.

[84] David Wetherall. “Active Network Vision and Reality: Lessions from a Capsule-based System”. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. Charleston, SC, USA, December 1999, pp. 64–79.

[85] David Wetherall and David Tennenhouse. “Retrospective on “Towards an Active Network Architecture””. In: *ACM SIGCOMM Computer Communication Review* 49.5 (November 2019), pp. 86–89.

[86] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”. In: *IEEE Open Architectures and Network Programming*. San Francisco, CA, USA, April 1998, pp. 117–129.

[87] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, and Arvind Krishnamurthy. “Runtime Programmable Switches”. In: *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, USA, April 2022, pp. 651–665.

[88] Zhaoqi Xiong and Noa Zilberman. “Do Switches Dream of Machine Learning? Toward In-Network Classification”. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. Princeton, NJ, USA, November 2019, pp. 25–33.

[89] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Characterizing Facebook’s Memcached Workload”. In: *IEEE Internet Computing* 18.2 (March 2014), pp. 41–49.

[90] Juncheng Yang, Yao Yue, and K. V. Rashmi. “A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter”. In: *ACM Transactions on Storage* 17.3 (August 2021), 17:1–17:35.

[91] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. “Using Trio – Juniper Networks’ Programmable Chipset – for Emerging In-Network Applications”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Amsterdam, Netherlands, August 2022, pp. 633–648.

[92] Fang Yu and Randy H. Katz. “Efficient Multi-Match Packet Classification with TCAM”. In: *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*. Palo Alto, CA, USA, August 2004, pp. 28–34.

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

[94] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. “OpenNetVM: A Platform for High Performance Network Service Chains”. In: *Proceedings of the workshop on Hot topics in Middleboxes and Network Function Virtualization*. HotMIddlebox ’16. Florianópolis, Brazil, August 2016, pp. 26–31.

[95] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. “ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining”. In: *Proceedings of the Symposium on SDN Research*. Santa Clara, CA, USA, April 2017, pp. 143–149.

[96] Peng Zheng, Theophilus Benson, and Chengchen Hu. “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*. Heraklion, Greece, December 2018, pp. 98–111.

[97] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. “Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection”. In: *arXiv:1904.08964 [cs]* (April 2019).

[98] Hang Zhu, Tao Wang, Yi Hong, Dan R. K. Ports, Anirudh Sivaraman, and Xin Jin. “NetVRM: Virtual Register Memory for Programmable Networks”. In: *Proceedings of*

the 19th USENIX Symposium on Networked Systems Design and Implementation. Renton, WA, USA, April 2022, pp. 155–170.

[99] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. “Application Defined Networks”. In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. Cambridge, MA, USA, November 2023, pp. 87–94.