Towards Offering Performance Guarantees in Software-Defined Networks

by

Amin Tootoonchian

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2015 by Amin Tootoonchian
Abstract

Towards Offering Performance Guarantees in Software-Defined Networks

Amin Tootoonchian
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2015

Software-Defined Networking (SDN) promises to bring the long-sought flexibility to networks by decoupling packet handling (data plane) and network control (control plane). SDN’s logically central control and poor performance of early SDN controller prototypes raised concerns about its performance and scalability. Moreover, the simple hardware-based datapath that legacy SDN assumed is a major obstacle to implement complex policies without significant performance penalty. In this thesis, I show that SDN could offer performance guarantees without sacrificing flexibility.

First, I present NOX-MT and HyperFlow that are controller designs that demonstrate the performance and scalability of that logically central control could offer. NOX-MT improves NOX’s (the legacy SDN controller) performance by an order of magnitude and can handle 1.6M events per second with 8 cores. HyperFlow transparently captures and replays state altering events to construct an eventually consistent global view across controllers. HyperFlow’s prototype can synchronize 1500 events across the network in a single RTT. Second, I propose an alternative SDN design that embraces the notion of edge-core distinction and software at the edge to free the network core from implementing complex policies and introduce the much-needed flexibility in data plane at the edge. Software datapaths are increasingly common in today’s networks but they introduce a new challenge: Performance guarantees that a software datapath provides in isolation no longer holds when it is consolidated with other software datapaths. Lastly, I show that
performance degradation under consolidation can be directly controlled (with less than 3% error) by isolating accesses to the last-level processor cache. I present a formulation of the service-level agreement enforcement problem the solution to which is a placement of software datapaths and their cache allocations that guarantees performance objectives.
Acknowledgements

My research benefited immensely from the contributions of many great people over the course of my graduate studies. First, I would like to thank my advisors, Prof. Yashar Ganjali and Prof. Scott Shenker, for their constant help and support. Yashar’s creativity, empathy, and upbeat character, and Scott’s unparalleled clarity, intuition, and wisdom are what any graduate student wishes for in an academic advisor. Scott made it possible for me to travel to Berkeley often and collaborate closely with the members of NetSys which made the last few years very fruitful and memorable. I would also like to thank the members of my advisory and thesis committee: Prof. Hans-Arno Jacobsen, Prof. David Lie, Prof. Guru Parulkar, Prof. Ding Yuan, and Prof. Ashvin Goel, for their valuable feedback, encouragements, and flexibility despite their busy schedules.

A meeting with Hans-Arno Jacobsen guided the design of HyperFlow presented in Chapter 2. The other part of that chapter is the product of a collaboration with Sergey Gurbonov, Martin Casado, Rob Sherwood, and Yashar Ganjali. Months of discussions with Martin Casado, Teemu Koponen, Aurojit Panda, Colin Scott, and Scott Shenker led to the work presented in Chapter 3. Katerina Argyraki, Sylvia Ratsanamy, and Scott Shenker contributed significantly to Chapter 4. Intel corporation provided us great support for this work, especially, Christian Maciocco and Andrew Herdrich of Intel Labs, and Priya Autee and Edwin Verplanke of Intel Network Processing Group without helped us every step along the way.

Research for my thesis was conducted at two labs: the Systems and Networking lab at University of Toronto, and NetSys group at University of California at Berkeley. The past and present members of the lab in Toronto: Geoff, Vivek, Ziad, Monia, Neda, Phillipa, Adam, Hossein, Alireza, Mohammad, Soudeh, Hamed, Milad, Sajad, Shiva, Soheil, Ioan, George, Nosayba, Larry, Andy, Bogdan, Suprio, Sahil, James, Peter(s), Daniel(s), Nilton, Stratis; and Berkeley: Panda, Colin, Murphy, Sangjin, Radhika, Justine, Chang, Yahel, Gautam, Peter, Ahmed, Ali, Vjeko, Shoumik, Saman. I am glad I got to spend my twenties in such great company. I would also like to thank the outstanding staff of the Department of Computer Science at UofT and International Computer Science Institute: Lynda Barnes, Jaclyn Considine, Jessica Craft, Lisa DeCaro, Marina Haloulos, Vinita Krishnan, Maria Eugenia Quintana, Joseph Raghubar,
Vladislav Sekulic. They each helped me tremendously when I needed it the most. And thanks to Suzanne Stevenson and Sven Dickinson for all the parties and the nights you took us to dinner and events.

Many great friends made these years much more enjoyable for me. I was very fortunate to make a great friend, Geoff Salmon, in just a few days after I arrived in Toronto. Aurojit Panda keeps me company whenever I am in Berkeley and is a very reliable, kind, and supportive friend. With Jackie Chi Kit Cheung, we explored the city, tried new things, and had so many unforgettable moments. Alireza Sahraei’s bold character was always a fun part of our lives. Milad Eftekhar, whom I have been friends with since we were 14, is kind, clever, and patient; thanks for choosing UofT over other schools. Bahar Aameri and Nezam Bozorgzadeh are the warm-hearted and compassionate friends who we always count on. Lalla, Siavosh, Siavash, Hanieh, Aditya, Orion, Eric, Niusha, Saman, Golnaz, Maryam, Yuval, Michael, Olga, Xuan, Nona, Mohammad, Mahnaz, Ilya, Giovanna, Joel, Tong, Jane, Wiebke, Erik, and many others who I am sure I forgot to list: many thanks for memorable parties, conversations, lunch/tea/coffee breaks, dinners, and board games.

And my family. I am forever grateful to my lovely parents (Maryam and Saeed) and sister (Negin) for all the unconditional love and support throughout my life. Roya and Morteza’s home was a home away from home for me in Toronto. I will forever remember Azadeh’s kindness, support, and all the thoughtful gestures. Finally and most importantly, I cannot express how grateful I am to Aida Nematzadeh. She is simply amazing and brilliant with a character full of love and empathy. After all these years, she finds a unique way to surprise me. She has supported me in hard times throughout these years and helped me grow personally, emotionally, and academically. Thanks Aida!
Contents

1 Introduction 1

2 Performance and Scalability of the Control Platform 10
   2.1 Improving the Control Platform Performance 11
       2.1.1 Building a High-Performance Controller 13
       2.1.2 Experiment Setup 18
       2.1.3 Controller Throughput 21
       2.1.4 Controller Response Time 26
   2.2 HyperFlow: A Distributed Control Plane for OpenFlow 31
       2.2.1 Design and Implementation 33
       2.2.2 Requirements on Controller Applications 38
       2.2.3 Evaluation 40
   2.3 Concluding Remarks 42

3 Evolving the Canonical SDN 43
   3.1 SDN’s Shortcomings 44
       3.1.1 SDN’s Need for Fabric 44
       3.1.2 Challenges of ASIC-driven Networking 48
   3.2 Overview 53
   3.3 Fabric 55
       3.3.1 Fabric Service Model 57
List of Tables

2.1 Worst-case controller response time .................................. 30
2.2 Controller response time as a function of the switch count ........... 30
2.3 HyperFlow message naming convention. ................................. 37
3.1 Fabric service model ................................................. 58
4.1 Test server specification. ............................................. 68
4.2 Workload characteristics ............................................ 71
4.3 Resource requirements to satisfy SLAs with last-level processor cache iso-
    lation .......................................................... 94
List of Figures

2.1 Best-case throughput with different number of threads. 22
2.2 Best-case throughput with different number of switches 24
2.3 Throughput with different number of threads and pending requests 25
2.4 Average maximum throughput for various levels of write-intensive workloads with 32 switches and 4 threads. 26
2.5 Response time CDF for different controllers with 32 switches, and $2^{12}$ pending requests. 28
2.6 Response time CDF for various maximum number of pending requests with 32 switches, and 4 threads. 29
2.7 A multi-site OpenFlow network with a single and multiple controllers 31
2.8 High-level overview of HyperFlow 34

3.1 Components of a network with fabric. 57

4.1 High-level architecture of the server with Intel Xeon E5-2658 v3 (Haswell series) processor. 69
4.2 Error of the throughput degradation method proposed in [23] 76
4.3 Variation in throughput with and without cache isolation 80
4.4 Sensitivity of different pipelines to memory controller contention 81
4.5 Cache sensitivity for various packet processing pipelines 83
4.6 Overhead of full last-level processor cache isolation. 85
Chapter 1

Introduction

Internet, and networking more generally, has transformed our lives in many different ways. While the Internet architecture has been undeniably successful, its infrastructure was not much so. The infrastructure has two components: (a) data plane (or datapath) that is in charge of processing and forwarding the traffic, and (b) control plane that controls the overall behavior of the network by updating the state of the datapath elements. The data plane is mostly hardware based, while the control plane is software based.\(^1\) It is widely agreed [55, 27, 51, 15, 59] that current networks are too expensive, too complicated to manage, too prone to vendor-lock-in, and too hard to change; these problems are commonly attributed to the rigidity of both the data and the control planes.

Today’s networks tightly couple these two planes together. Each datapath element comes bundled with a full-blown control plane that implements a set of control protocols and runs on a low-powered management CPU with very little memory. In this setup, each new control protocol is either proprietary to a vendor (results in vendor lock-in), or must be standardized (a very slow process) to work across equipment of different vendors. Also, from a design and implementation standpoint, the legacy control plane lacks any notion of modularity and layering: Control protocols most often have their

\(^1\)A hardware-based datapath element is one that uses purpose-built hardware (e.g., application-specific integrated circuit or ASIC) to process packets. The control logic for a software-based control element is implemented as a program that runs on a commodity CPU (e.g., an x86 CPU).
own low-level mechanisms to carry out common tasks such as discovery, state collection and dissemination, and failure detection and recovery. The hardware vendors have for long, admirably so, managed the complexity of implementing new control protocols in this highly distributed and low resource setting. Besides these complexities, this model inevitably puts a single entity – the hardware vendor – in charge of everything. No third party or network operator can add new control functionality without vendor’s support.

The state of affairs in the data plane is not inspiring either. The datapath’s reliance on purpose-built non-programmable hardware is a significant barrier for adding new functions. The time it takes for a new hardware design to reach the market is very long (a year and a half). But more importantly, many ideas do not find their way to the datapath in the first place. This led to middleboxes becoming the norm over the past decade [74]. Middleboxes are intermediary datapath elements that implement new packet processing functions such intrusion detection and prevention, firewall, network address translation, load balancing, caching, and WAN optimization. Even though middleboxes bring the long sought flexibility to the datapath, deploying and provisioning them remains difficult. They have to be carefully placed in the network and traffic should be correctly steered towards them.

There has been many attempts at addressing these issues over the past few decades [70, 85, 31]. The new wave of such efforts are centered around a few paradigms, most notably Software-Defined Networking (SDN) [55, 32, 51, 64] and Network Function Virtualization (NFV) [59].

SDN derives from a long line of research (see [31, 10, 14, 13] for a small sampling). It decouples network control from datapath elements (e.g., switches and routers) and offers a logically centralized way to control the network. The network control platform (i.e., the network operating system [32]) manages datapath elements using a standard interface (e.g., OpenFlow [55]). In turn, it provides an interface for control application to access the global view and modify the network state as desired. But, in essence, the role of
the SDN control platform is to implement a set of basic primitives that are common to most control applications such as discovery, state collection and dissemination, and fault tolerance. Control applications in this new setting can directly implement the intended logic without the need to directly address these complex and low-level details.

Perhaps most fundamentally, SDN was an attempt to provide a clean set of abstractions for the control plane, much the same way layering does for the data plane [73]. These abstractions enable us to build scalable, flexible, and modular network control systems, and greatly improve the clarity and quality of network control planes. This is evident from recent research on debugging [71], verification [46, 47], policy enforcement [13], simplifying programming [28, 66].

SDN has a fully general control plane but a constricted datapath. OpenFlow, in particular, abstracts away the datapath as a set of match/action tables with limited flexibility in terms of the supported matches and actions. But, today’s networks are replete with middleboxes that perform a wide variety of functions, ranging from the generally applicable (e.g. firewalls, intrusion detection and prevention) to the extremely specific (e.g. media gateways [16, 26], broadband remote access servers) which does not fit in the SDN model of simple data plane. NFV, on the other hand, focuses on datapath flexibility.

NFV offers to replace special-purpose single-function hardware devices with virtualized network functions (VNFs), typically running as a VM, that could run on racks of generic servers. With recent advancements in software packet processing stack (e.g., Intel’s DPDK [38] and netmap [68]) and commodity processors (e.g., Intel’s data direct I/O [20], integrated memory and PCIe controllers), software-based packet processing are performant enough to be viable. The NFV approach offers many advantages, such as low cost, ease of deployment, elasticity and consolidation, faster provisioning of services, and statistical multiplexing over a shared infrastructure (e.g., racks of servers).

SDN’s flexibility in network control and NFV’s flexibility in datapath, together, have
the potential to address the above-mentioned network infrastructure shortcomings. But it also introduces many new challenges.

It is common for networks to offer service-level agreements (SLAs) with guarantees on different performance metrics such as up time, bandwidth, and delay. Middleboxes, if present, should be provisioned for the worst case due to difficulty of dynamically scaling with demand. Essentially, networks today are vertically integrated with individual elements provisioned to provide an end-to-end performance guarantee. It is not, however, immediately clear how network infrastructures based on SDN and NFV could provide similar end-to-end performance guarantees. Below, we outline few hurdles that these networks face.

Early SDN controller prototypes [13, 32] demonstrated a level of performance well beyond the requirements in numerous settings (such as homes [94, 80] and the enterprise [13]), but later studies of large networks [44, 6] raised concerns on whether SDN is suitable for such environments: To handle the peak control traffic in some of the studied networks [6], even with complete sharding [82], more controllers need to be deployed than switches and routers. Some works [95, 19] address these performance concerns by keeping all or most of traffic in the datapath. DIFANE [95], for instance, precomputes all forwarding state and cleverly tries to fit this state in the limited datapath memory. Such designs, however, are constrained to control policies that could be implemented using the scarce datapath flow table capacity.

Surprisingly, the SDN’s choice of general control plane but simple datapath may make it very difficult to reason about network performance. It is not always possible to directly implement a high-level policy using with the limited functions and resources in datapath. Traditionally, special-purpose datapath elements (e.g., middleboxes) were deployed to add the missing functions, but with SDN it is very luring to implement

---

2An SLA refers to a service contract between a provider and a user where the service is defined and its level of quality is agreed upon. In this thesis, we focus on the performance aspects of network services (e.g., throughput and latency) offered by a network provider.
missing functions in the control plane. As soon as we make the choice to conflate packet processing with network control, among other complications that arises, network performance is unnecessarily constrained by the limited control bandwidth and processing capacity of the control plane. SDN also makes it easy to conflate packet delivery with other network functions. While this may even lead to better performance with on-path packet processing, it becomes considerably harder to provision resources for packet delivery and other functions independently especially considering that their failures are not independent anymore.

NFV’s software-based approach, on the other hand, brings an unparalleled flexibility to datapath. But there is one oft-overlooked disadvantage to the move to software. Because they had fixed functionality and dedicated single-purpose hardware, the physical incarnations of these functions had well-understood performance specifications and QoS-related knobs, which allowed operators to offer service-level agreements (SLAs) to their customers.\(^3\) Once these functions move to software, providing such performance specifications is much harder. The difficulty in predicting comes largely from the fact that such VNFs would coexist on a single shared infrastructure; even if it is understood how each VNF performs in isolation, one could not predict how they would perform when colocated on the same machine.

Undeniably, the flexibility that SDN and NFV introduce in the control and data planes are fundamental to network evolution [65]. In this thesis, we show that this flexibility could be had without the loss in ability to offer performance guarantees. In a nutshell, we address the above challenges as follows:

- We show that, fundamentally, SDN’s logically central control is neither a scaling nor a performance bottleneck (Chapter 2).

- We sidestep the SDN performance pitfalls with an alternative design that introduces

\(^3\)An SLA refers to a service contract between a provider and a user where the service is defined and its level of quality is agreed upon. In this thesis, we focus on the performance aspects of network services (e.g., throughput and latency) offered by a network provider.
flexibility and modularity into the datapath (Chapter 3).

- We propose a method to isolate the performance of colocated software packet processing applications (i.e., VNFs).

First, we look at the control plane performance and scalability issues (see Chapter 2). We present NOX-MT [88] that with sustains nearly 400k requests per second per core with a maximum response time of 2ms on a (7-year-old) 2GHz processor (compared to throughput of 30k requests per second and latency of 10ms of NOX [32]).\(^4\) It scales well with limited number of cores and enough parallelism in the workload: It can sustain a 1.6M requests per second throughput and sub-2ms latency with eight processing cores. This implies that at this network-wide load level and with a single controller the time it takes an SDN controller to react to a network event is at most 2ms worse than that of a local controller.\(^5\) A significantly lower latency could be achieved by operating at a lower utilization and deploying more controllers to handle the aggregate load.

Complementary to NOX-MT [88], HyperFlow [87] helps scale the control platform while maintaining the global view abstraction transparent from the control applications (see Chapter 2).\(^6\) HyperFlow runs on each controller and synchronizes the global view across them. To do so, it captures the events observed on each controller and replays them on others. It relies on applications to gracefully handle slightly reordered network events. This design scales very well for read-intensive workloads where most of the (non-state-altering) events are filtered locally. As the rate of state-altering events increases the time it takes for the views to converge increases. To gracefully handle this transient inconsistency in views, HyperFlow makes each controller authoritative for the events generated by the switches directly connected to that controller. Only the updates from the authority controller can reach the datapath and every other update is filtered by HyperFlow.

\(^4\)This study first appeared in [88].
\(^5\)This assumes that the control network is well provisioned.
\(^6\)HyperFlow first appeared in [87].
Chapter 1. Introduction

HyperFlow and NOX-MT show that, with careful design, SDN’s logically central control does not inherently limit the network performance. But the more pressing issue remains: SDN allows designs in which it is very difficult to reason about performance and offer guaranteed performance. The two high-level issues are the conflation of packet delivery with more complex network functions and conflation of packet processing and control plane processing.

To sidestep these performance pitfalls, we present an alternative SDN design based on fabric [15] and software edge (see Chapter 3). Fabric introduces functional modularity in the data plane by distinguishing between the function of the network core and edge. The network core is in charge of packet delivery while all other network functions are implemented at the edge. The benefits of this distinction is two-fold: first, it enables us to provision for packet delivery independently of other network functions; second, the core can be simple and future proof supporting a wide range of new functionality at the edge (the same way that Internet’s end-to-end architecture spurred innovation). To mitigate the challenges of implementing new functions on constricted hardware (both functionality and resource-wise), our design uses software at the edge for implementing new network functions. Disaggregation at the edge along with the possibility of elastic network functions, make software datapath at the edge feasible and desirable. We emphasize that software edge does not preclude use of hardware at the edge. It could benefit from hardware assist when available (e.g., acceleration of a function by a coprocessor, NIC, FPGA, etc.) or even software processing could be completely bypassed if not necessary. Finally, we note that both the edge/core distinction (e.g., MPLS [70]) and use of software packet processing (e.g., VMware NSX [60], Open vSwitch [64], NFV) are common in today’s networks. We are not claiming any novelty for either, but pointing out that SDN should evolve to embrace them.

With the above design, the individual functions at the edge and the core could be

\footnote{Fabric first appeared in [15].}
independently provisioned; but consolidation of network functions on a single machine at the edge introduces a new challenge. Even if network functions are run on independent core on a single machine, they may affect each other quite significantly because they share a few resources (e.g., last-level CPU cache, memory controller, I/O controller) [23].

Previous work [23] provides some hope: it proposes a simple method to predict how packet processing pipelines’ performance is affected under contention for shared resources. Unfortunately, with the new hardware and software stacks the method has much lower accuracy than previously shown (see Chapter 4). Also, the prediction is dependent on the knowledge of (or correctly predicting) the configuration and input traffic pattern of all colocated pipelines.

We show that, for a wide range of packet processing pipelines that we studied, isolating the last-level processor cache (available on Intel Haswell Xeon processors), among other shared resources, is sufficient to isolate the performance of consolidated software datapaths. Therefore it is possible to accurately (less than 2% error) predict the pipelines performance given its allocated last-level cache size, and expected configuration and input traffic pattern. As such, any wild changes in a pipelines configuration or traffic patterns does not violate the performance guarantees that the colocated pipelines are provided with. We present a sample formulation to the SLA enforcement problem as an integer program the solution to which specifies number of instances and placement for each pipeline, as well as their cache allocations.

Chapter 2 looks at SDN controller performance and scalability: it presents HyperFlow [87], NOX-MT and a comparative study of single-node controller performance using microbenchmarks [88]. In Chapter 3, we present our alternative SDN design with a discussion of fabric interfaces and software edge. Chapter 4 studies the predictability of consolidated software packet processing pipelines: it shows that previous work does not address the problem sufficiently, and that for a wide range of packet processing functions we looked at, isolating last-level cache (now part of commodity processors featureset) is
enough to guarantee performance isolation.
SDN’s logically central network control brought up concerns about its performance and scalability. Different studies [82, 44, 6] underscored the need for distributed control platforms to handle data center traffic. The large disparity between the performance of SDN controllers motivated a spate of interesting work, most notably DevoFlow [19] and DIFANE [95], that mitigate this issue by carefully programming the dataplane to keep all/most of traffic from reaching the controller. Our work presented in this chapter, in contrast, looks at the performance and scalability of the control platform itself.

In Section 2.1, we present NOX-MT (a performant multi-threaded successor to NOX) and, with microbenchmarks, studies the performance characteristics of SDN controllers.\footnote{This work appeared in [88].} To the best of our knowledge, NOX-MT was the first effort in enhancing controller performance and motivated other controllers to improve. We emphasize that our contribution is to show that SDN performance is not fundamentally limited due to poor controller performance and that the SDN controllers can be optimized to be very fast. A single NOX-MT instance handles 200k events per second using a single core and 1.6M events...
per second with 8 cores. Even though this may be sufficient to handle the control load of a wide range of network policies for mid-sized networks, it is crucial to show that SDN’s central control scales to larger networks through deployment of multiple controllers.

HyperFlow (Section 2.2) is a distributed OpenFlow control plane that, transparently from control applications, maintains an eventually consistent global network view across controllers.\(^2\) To reconstruct this view, HyperFlow captures state-altering events on each controller and replays them on others. Therefore, to ensure state consistency, control applications must tolerate event reordering (which is a common trait of legacy control planes). HyperFlow’s design scales for policies in which only a small fraction of network events alter the global view. Our prototype synchronizes 1500 network events across the controllers within a single RTT. Even though there is a lot of room for improvements in our prototype, for policies whose global view may consist of a much larger of number of events, HyperFlow may not be suitable. An alternative design for such policies is to shard the control state across control domains (akin to OSPF areas) and use HyperFlow design within each area to synchronize state.

### 2.1 Improving the Control Platform Performance

While it has been argued that SDN is suitable for some deployment environments (such as homes [94, 80], data centers [3], and the enterprise [13]), delegating control to a remote system has raised a number of questions on control-plane scaling implications of such an approach. Two of the most often voiced concerns are: (a) how fast can the controller respond to data path requests?; and (b) how many data path requests can it handle per second?

There are some references to the performance of SDN systems in the literature [93, 13, 10]. For example, an oft-cited study shows that a popular network controller (NOX)\(^2\)This work first appeared in [87].
handles around 30k flow initiation events\(^3\) per second while maintaining a sub-10ms flow install time [82].

Unfortunately, recent measurements of some deployment environments suggests that these numbers are far from sufficient. For example, Kandula et al. [44] found that a 1500-server cluster has a median flow arrival rate of 100k flows per second. Also, Benson et al. [6] show that a network with 100 switches can have spikes of 10M flows arrivals per second in the worst case. In addition, the 10ms flow setup delay of an SDN controller would add a 10\% delay to the majority of flows (short-lived) in such a network.

This disconnect between relatively poor controller performance and high network demands has motivated a spate of recent work (\textit{e.g.}, [19, 95]) to address perceived architectural inefficiencies. However, there really has been no in-depth study on the performance of a traditional SDN controller. Rather, most published results were gathered from systems that were never optimized for performance. To underscore this point, as we describe in more detail below, we were able to improve the performance of NOX, an open source controller for OpenFlow networks, by more than 30 times.

Therefore, our goal is to offer a better understanding of the controller performance in the SDN architecture. The specific contributions are:

- We present \textit{NOX-MT} a publicly-available multi-threaded successor of NOX [32]. The purpose of NOX-MT is to establish a new lower bound on the maximum throughput. Unlike previous studies and implementations that were not tuned for performance, NOX-MT uses well-known optimization techniques (\textit{e.g.}, I/O batching) to improve the baseline performance. These optimizations lets NOX-MT outperform NOX by a factor of 33 on a server with two quad-core 2GHz processors.

- We design a series of flow-based benchmarks embodied in our tool, \textit{cbench}, that we make freely available for others use. Cbench emulates any number of OpenFlow switches to measure different performance aspects of the controller including the

\[^3\text{In the rest of this chapter, we use flow initiation and requests interchangeably.}\]
minimum and maximum controller response time, maximum throughput, and the
throughput and latency of the controller with a bounded number of packets on the
fly.

- We present a study of SDN controller performance using four publicly-available
OpenFlow controllers: NOX, NOX-MT, Beacon, and Maestro [11]. We consider
NOX as the baseline for our performance study since it has been previously used
in different papers [82, 95, 19].

2.1.1 Building a High-Performance Controller

Ideally, a flow-based SDN control plane should have: (a) high flow setup throughput
minimizing the number of required controllers to handle a specific flow initiation rate;
and (b) small response times (with low variation) to minimize the overhead on flow
completion time.

NOX [32] – whose measured performance motivated several recent proposals on im-
proving control plane efficiency – has a low flow setup throughput and a high flow setup
rate, however. Fortunately, this is not an intrinsic limitation of SDN: NOX is not op-
timized for performance, and is single-threaded not utilizing all the available compute
power on multiprocessor machines.

To better understand SDN control plane performance, we present NOX-MT, an op-
timized multi-threaded successor of NOX. NOX-MT outperforms NOX by a factor of 33
on a server with two quad-core 2GHz processors. Additionally, on the same machine in
a single-threaded run, NOX-MT performs six times better than NOX.

The techniques we employed to optimize NOX are quite well-known including: I/O
batching to improve I/O handling, porting the I/O handling harness to Boost asyn-
chronous I/O (ASIO) library which simplifies multi-threaded operation, and using multiprocessor-
aware malloc implementation alternatives which scale linearly with the number of pro-
cessors.
We note that NOX-MT only provides a new lower-bound on the maximum achievable controller throughput. There are yet many performance deficiencies available in NOX-MT’s code base than we plan to address in future including but not limited to: heavy use of dynamic memory allocation and redundant memory copies on a per-request basis, and using locking were robust wait-free alternatives exist. Addressing each of these issue would have a significant impact on the control plane performance. In what follows, we describe NOX’s operation model, its performance bottlenecks, and our modifications to improve it.

**NOX Internals**

NOX is an event-driven network controller. Applications running atop NOX register to listen for different event types. NOX’s event dispatcher module keeps track of these registrations in a per-event list sorted by the handler priority. Upon receipt of each OpenFlow message, NOX’s message processing routine converts it to an event corresponding the OpenFlow message type. The event is then dispatched through the event dispatcher module.

The minimum NOX response time (over LAN) is 140 microseconds with a very small variation (e.g., due to interrupt coalescing in the NIC card). However, it can only handle around 60\(k\) flow initiations per second (i.e., 40 Mbps of control traffic considering 82-byte-sized packets we used in our experiments) for which delays are in the order of several milliseconds (due to huge amount of buffering available in the system).

**Single-threaded Performance Bottlenecks**

**I/O Handling:** NOX polls OpenFlow connections in a loop continuously for messages, reading at most one message at a time. Similarly to serving requests, NOX sends commands/replies to switches one at a time. This translates into a large number of read/write socket system calls each with a very small buffer (typically less than 131 bytes). This
incurs a notable overhead when the controller is under high loads for three reasons.

First, under high loads NOX ends up issuing a huge number of costly read/write system calls which is proportional to the number of requests. Second, serving requests individually causes each request to turn into a task in the task pool. With a large number of tiny tasks, the overhead of locking required to guard concurrent access to the task queue outweights the computation done serving the tasks. Therefore, to mask the overhead of thread synchronization, we ideally would want have fewer but longer tasks. Third, reading from and writing into buffers smaller than 128 bytes is almost as costly as page-sized buffers, so operating with bigger chunks of data at once turns into notable savings.

Our solution is to batch requests and replies together where possible. Each socket’s asynchronous receive operation reads as many bytes as available in the buffer, and its completion handler serves all the requests sequentially in the same context. We note that this batching is on a per-connection basis, and with larger number of connections I/O overhead would be larger. Finally, the way we implement I/O batching never incurs any extra overhead on the response time: we only batch writes when the socket is busy, otherwise we flush the request directly to socket to avoid any extra delays.

**Dynamic Memory Allocation:** For each OpenFlow message received, NOX dynamically allocates small buffers in several places. Under high loads this becomes increasingly costly and must be avoided to the extent possible. Ideally we should use memory pools with constant time memory allocation. However, we have left it for future work, but our preliminary results suggest that this change would significantly improve the performance. Meanwhile, we experimented with Google’s TCMalloc [30] which is optimized for multi-threaded operation. Our results suggest that TCMalloc improves the controller throughput by 20%.
Enabling Multi-threaded Operation

An SDN controller is a perfect candidate for multi-threading for two reasons. First, almost all the controller tasks are CPU bound. Second, the control plane workload consists of mostly read-only requests with infrequent updates to shared data structures. So, we expect to see a near-linear speedup with respect to the number of processors. However, NOX is effectively single-threaded: all threads in NOX are cooperatively scheduled on a single processor.

Our first attempt to enable multi-threaded operation in NOX was to add worker threads, each in their own thread group, all concurrently running the main polling loop. The main polling loop simply iterates over pollable services (i.e., event and timer dispatchers, and individual OpenFlow connections) and serves the outstanding requests if any. In each iteration, a running thread dequeues a pollable service from the pollables queue, polls and serves it, and finally pushes it back in the pollables queue in which the queue is protected with a lock during enqueue and dequeue operations. We had to modify several libraries, including NOX’s cooperative threading library to protect all the shared data structures. Even though this approach showed an improvement of a factor of five with eight threads, we decided not to continue with it because: (a) the process was overly complicated and error-prone; (b) it was quite hard to apply the rest of the optimizations discussed here; and (c) reasoning about the controller performance was quite hard. In short, we needed to simplify the codebase.

Our solution to this issue was to rewrite core components of NOX to use Boost’s Asynchronous I/O (ASIO) library, a cross-platform C++ library for network and low-level I/O programming that provides a consistent asynchronous model. For that, we removed all the dependencies on the NOX cooperative threading and asynchronous I/O libraries; then updated the polling loop and pollable services to work with ASIO. These

---

4In the cooperative threading model, the programmer explicitly (by calling yield) or implicitly (by blocking) instructs the thread scheduler to run the next thread in the thread group. In NOX, all threads are in the same group, removing the hassle of protecting shared resources.
modifications significantly simplified the NOX codebase and removed the need for several core NOX libraries (more than 6000 lines of code).

ASIO greatly simplifies multi-threading. Upon boot-up, NOX-MT simply spawns worker threads to poll the I/O service object.\(^5\) In ASIO, asynchronous operations are posted to an I/O service object along with a completion handler that is to be invoked once the operation has completed (\textit{e.g.}, completion of an \texttt{async\_read} operation on a network socket). The invocation takes place in the context of one of the threads polling the respective I/O service object.

Moreover, ASIO evenly distributes the compute load among all the worker threads. At the same time we keep handling individual requests of each switch local to a single processor to avoid cross-core synchronization overhead. For optimal operation, the number of worker threads should not exceed the number of available cores. That is because NOX-MT does not block to serving any request, so having more threads than cores only adds to the contention for serving compute tasks.

\textbf{Takeaways}

It is not surprising that the above optimization techniques are closely related to those used by software packet processors: batch packets to amortize overhead, share nothing when possible, avoid dynamic memory allocation, and run to completion on a single core. The network control workloads at the time of this study were mostly similar to packet processing workloads. The significant performance gap between NOX-MT and highly-optimized software packet processors is evidence that a lot remains to improve. In retrospect, perhaps the most important lesson is that latency-sensitive network control platforms should only use primitive functions stripped down to their essence and have very simple execution paths.

\(^5\)ASIO is thread safe and protects concurrent polling of the I/O service object with an internal lock.
2.1.2 Experiment Setup

In an effort to quantify controller performance, we create a custom tool, *cbench* [76], to measure the number of flow setups per second that a controller can handle. In SDN, the OpenFlow controller must setup and tear down flow-level forwarding state in OpenFlow switches. This “flow setup” process can happen statically before packets arrive (“proactively”) or dynamically as part of the next hop lookup process (“reactively”). Reactive flow setups are particularly sensitive because they add latency to the first packet in a flow. Once set up, the flow forwarding state remains cached on the OpenFlow switch so that this process is not repeated for subsequent packets in the same flow. The OpenFlow controller also communicates how long to cache the state: either indefinitely, after a fixed timeout, or after a period of inactivity. We choose to focus on the flow setup process because both because it is integral to SDN and because it is perceived to be the likeliest source of performance bottleneck.

Our tool, *cbench* [76], measures various performance issues related to flow setup time. Cbench emulates a configurable number of OpenFlow switches that all communicate with a single OpenFlow controller. Each emulated switch sends a configurable number of new flow (OpenFlow packet_in) messages to the OpenFlow controller, waits for the appropriate flow setup (OpenFlow flow_mod or packet_out) responses, and records the difference in time between request and response.

Cbench supports two modes of operation: latency and throughput mode. In latency mode, each emulated switch maintains exactly one outstanding new flow request, waiting for a response before soliciting the next request. Latency mode measures the OpenFlow controller’s request processing time under low-load conditions. By contrast, in throughput mode, each switch maintains as many outstanding requests as buffering will allow, that is, until the local TCP send buffer blocks. Thus, throughput mode measures the

---

6While the specifics of this description use an OpenFlow-based terminology, the flow setup operation is common to all flow-based network architectures, e.g., setting up virtual circuits in ATM, configuring labels in MPLS, or physical circuits in optical networks.
maximum flow setup rate that a controller can maintain. Cbench also supports a hybrid mode with \( n \)-new flow requests outstanding, to explore between these two extremes.

We analyzed cbench to ensure that it is not a bottleneck in our experiments. For that, we instrumented cbench to report the average number of requests on the fly for different experiments. Cbench also reports average throughput and response time. According to Little’s theorem \( (L = \lambda W) \) [52] the average number of outstanding requests must match the product of the system throughput and the average response time. Throughout our experiments these numbers were in agreement. The slight differences are an artifact of taking the average over the samples collected in each run.

Using cbench, we evaluated the flow setup throughput and latency of four publicly available OpenFlow controllers using cbench: (a) NOX [32] is a single-threaded C++ OpenFlow controller adopted by both industry and academia. (b) NOX-MT is an optimized multi-threaded successor of NOX we developed and presented in this paper. (c) Maestro [11] is a multi-threaded Java-based controller from Rice university. (d) Beacon\(^7\) is a multi-threaded Java-based controller from Stanford university and Big Switch Networks. We used the latest available version of each controller available as of May 2011.

In all the experiments, each controller runs the L2 switching application provided by the controller.\(^8\) For each switch on the path, the switch application performs MAC address learning. Each packet is forwarded out of the last port on which the traffic from the destination MAC address is seen. Packets with unknown destinations are flooded. In NOX, for each switch, the mapping between the MAC-switch tuple and the port number is stored in a hash table. The switch application has mostly read-only work load: only

\(^7\)Through private correspondence with Beacon’s author, we were informed that Beacon performs twice better on a 64-bit OS. We note that the results reported in this chapter are for 32-bit runs. However, we are not aware of the reason for this gap in performance.

\(^8\)We choose to use L2 switching for our evaluation for two primary reasons. First, it provides a lower-bound for lookup since the full forwarding decision can be accomplished with a hash followed by a single lookup. Thus, the test is not heavily augmented by overhead of the lookup (which could be the case with something like longest-prefix match implemented in software). And secondly, basic switching has been implemented in all the controllers we tested.
requests with newly observed source MAC addresses trigger an insert/update in the hash table, and the number of such events is proportional to the product of the number of hosts in the network and the number of switches.

To minimize interference, we run the controller and cbench on two separate servers (8 \times 2GHz and 4 \times 2.13GHz CPU cores respectively, both with 4GB of DDR2 ram). All servers run Debian Squeeze with Linux 2.6.32-5-686-bigmem, gcc 4.4.5, GNU libc 2.11, GNU libC++ 3, Boost 1.42, Sun Java 1.6, and TCMalloc 1.5. NOX and NOX-MT are compiled with no support for python and with debugging disabled. All controllers are run with logging disabled and no verbose output. For NOX, NOX-MT, and Maestro threads are bound to distinct CPUs. We used TCMalloc [30] malloc implementation since it provides a faster alternative to GNU libc’s malloc with better scalability in multi-threaded programs. Also, unless otherwise noted, we run the experiments with default sizes for Linux networking stack buffers as well as NIC drivers’ ring buffers. Cubic is the default congestion control algorithm on Linux 2.6.32. Each server has two Gigabit ports directly connected to the other server. The Gigabit links are teamed together to provide a 2Gbps control bandwidth required for some experiments. Throughout our experiments cbench’s CPU utilization is consistently less than 50%. We occasionally run multiple parallel instances of cbench on different processors to verify cbench’s fairness in serving different emulated switches (sockets) as well as its accuracy.

Each test consists of 4 loops each lasting 5 seconds. The first five seconds (first loop) is considered as controller warm-up and its results are discarded. Each test uses 100k unique MAC addresses (representing 100k emulated end hosts). For experiments with fixed number of switches, we chose to present the results for 32 emulated switches because we do not expect a large number of switches to be stressing the network simultaneously. To maximize the stress on the controller and to ensure that the bandwidth is not the bottleneck to the extent possible, we use 82-byte sized OpenFlow packet-in messages
2.1.3 Controller Throughput

Individual controllers’ throughput is an important factor in deciding the overall number of controllers required to handle the network control load. Our focus in this section is to study the maximum throughput in the system in various settings.

**Maximum throughput:**

Figure 2.1 shows the average maximum throughput of different controllers with different number of threads. The results suggest that: (a) NOX-MT shows a significantly better performance compared to the other controllers. It saturates 1Gbps control bandwidth with four 2GHz CPU cores. (b) As expected, all the multi-threaded controllers achieve near-linear scalability with the number of threads (cores), because the controller’s workload is mostly read-only minimizing the amount of required serialization.

**Relation with the number of active switches:**

Ideally, controller’s aggregate throughput should not be affected by the number of switches connected to it. However, increased contention across threads, TCP dynamics, and task scheduling overhead within the controller are factors that can lead to a degraded performance if we have a large number of highly active switches.

To study the impact of the number of switches on controller performance, we measure the average maximum throughput with different number of switches and threads in Figure 2.2. We observe that controllers’ are optimally utilized when the number of switches served by each thread is bounded; e.g., NOX-MT’s thread should each handle between one and eight switches. Handling more switches than this threshold reduces the overall

---

9 All our estimates on the bandwidth usage is based on this size. For instance, throughout the paper we map 1.45M requests to 1Gbps control bandwidth. That is because \( \frac{1.45 \text{Mreq}}{\text{sec}} \times 82 \text{bytes req} \times \frac{8 \text{bits}}{\text{byte}} = 951.2 \text{Mbps} \) and assuming that the majority of Ethernet frames are MTU-sized (to account for Ethernet+IP+TCP overhead), this number roughly corresponds to 1Gbps.
Figure 2.1: Average maximum throughput achieved with different number of threads. We use 32 emulated switches and 100k unique MAC addresses per switch. NOX-MT saturates 1Gbps (1.45Mreq/sec) of control bandwidth using four 2GHz CPU cores. Since NOX is single-threaded it only has a single point in this and similar graphs.
throughput, because I/O handling overhead and contention on the task queue and other shared resources increases, and I/O and job batching become less effective.

We note that the number of switches presented in our experiments only reflects the highly active (i.e. producing an extremely large number of requests per second) switches. In a typical network it is very unlikely that all the switches are highly active at the same time, so each controller should be able to manage a far larger number of switches.

**Relation with the load level:**
Cbench’s throughput mode effectively keeps the pipe between itself and the controller full all the time. However, since we have large buffers across all layers in the networking stack (e.g., network adapter’s ring buffer, network interface’s send and receive queues, TCP buffers, etc.), this pipe is quite large.

Our experiments verify that average maximum throughputs with $2^{12}$ outstanding requests are very close to the ones with unlimited number of outstanding requests (see Figure 2.3). With the same throughput, it follows from the Little’s law that doubling the number of requests in the system doubles the response time. In our experiments, the average delay with unlimited outstanding requests is almost an order of magnitude larger than the limited ones even though they achieve the same level of throughput. We study the controller response time corresponding to different load levels in Section 2.1.4.

**Effect of write-intensive workload:**
Write-intensive workloads increase the contention in the network control applications. For the switch control application, having a large number of unique source MAC addresses result in a write-intensive workload. As shown in Figure 2.4, both Maestro and Beacon are significantly affected by this workload. However, NOX-MT does not exhibit a similar behavior. That is because NOX-MT’s switch application minimizes contention in such scenarios by partitioning the network’s MAC address table among a pool of hash tables selected by the hash of the MAC address.
Figure 2.2: Average maximum throughput with different number of switches. Having more threads than the number of switches does not improve throughput. NOX-MT shows nearly identical average maximum throughput for 16, 32 and 64 emulated switches. With 256 emulated switches the performance of all three controllers degrades.
Figure 2.3: Average throughput with different number of threads and limits on the maximum overall number of requests on the fly. It is possible to achieve an almost maximum throughput even with limited number of requests on the fly for all three controllers.
2.1.4 Controller Response Time

In a software-defined network where flow setup is performed \textit{reactively}, controller response time directly affects the flow completion times. In this section, we present benchmarks to measure minimum (least load) and maximum controller (maximum load) response time of SDN controllers. Then, we study the relation between the load level and response time as well as the number of switches and response time.

**Minimum response time:**
To measure minimum control plane response time, we constrain the number of packets on the fly to be exactly one. Besides controller service time, this minimum response time includes traversal of networking stacks on both the controller and cbench sides twice, as well as the processing time of the controller. Average response times of all controllers are between 100 and 150 microseconds.

**Maximum response time:**
Maximum response time for each controller is observed when the maximum number of packets on the fly is not bounded \((i.e.,\) when the benchmakrer exhausts all the buffers in between the emulated switch and the controller). Since NOX-MT has the highest throughput, it has the least response time compared to others. As we see in Table 2.1, the average number of packets on the fly across these experiments is inversely proportional to the controller throughput (in accordance with the Little’s law).

**Relation with the load level:**

To better understand the relation between controller load and response time, we plotted the response time CDFs fixing the load level to \(2^{12}\) requests on the fly (see Figure 2.5) and the response time varying the load level (see Figure 2.6). We find that for the same workload adding more threads decreases the response time. Also, doubling the number of outstanding requests doubles the response time, but does not significantly affect the throughput (see Section 2.1.3).

**Relation with the number of active switches:**

Varying the number of active switches (see Table 2.2), we observe the same pattern for delay as we observed for throughput in Figure 2.2. Adding more CPUs beyond the number of switches does not improve latency, and serving far larger number of switches than available CPUs results in a noticeable increase in the response time.
Figure 2.5: Response time CDF for different controllers with 32 switches, and $2^{12}$ pending requests.
Figure 2.6: Response time CDF for various maximum number of pending requests with 32 switches, and 4 threads.
Table 2.1: Worst-case average response time and standard deviation (milliseconds) for various number of threads with 32 switches and unlimited number of outstanding flow setup requests.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOX</td>
<td>631.87 ± 44.62</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NOX-MT</td>
<td>349.44 ± 127.45</td>
<td>143.55 ± 63.19</td>
<td>92.59 ± 42.40</td>
<td>66.34 ± 38.32</td>
</tr>
<tr>
<td>Beacon</td>
<td>1028.51 ± 175.32</td>
<td>634.83 ± 204.99</td>
<td>394.21 ± 205.59</td>
<td>293.80 ± 233.33</td>
</tr>
<tr>
<td>Meastro</td>
<td>1268.58 ± 84.38</td>
<td>783.56 ± 56.56</td>
<td>558.40 ± 338.04</td>
<td>361.01 ± 301.68</td>
</tr>
</tbody>
</table>

Table 2.2: Response time (milliseconds) varying the number of switches for runs with 4 threads and $2^{12}$ requests on the fly.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOX-MT</td>
<td>9.92 ± 6.39</td>
<td>3.80 ± 0.69</td>
<td>3.51 ± 1.10</td>
<td>3.84 ± 2.51</td>
<td>4.63 ± 6.65</td>
<td>8.63 ± 18.73</td>
</tr>
<tr>
<td>Beacon</td>
<td>30.47 ± 14.99</td>
<td>14.85 ± 18.69</td>
<td>11.89 ± 26.22</td>
<td>11.86 ± 35.69</td>
<td>18.74 ± 85.57</td>
<td>30.48 ± 116.02</td>
</tr>
<tr>
<td>Meastro</td>
<td>22.75 ± 10.55</td>
<td>15.98 ± 10.18</td>
<td>15.97 ± 11.08</td>
<td>26.09 ± 43.76</td>
<td>23.35 ± 49.69</td>
<td>29.84 ± 57.88</td>
</tr>
</tbody>
</table>
2.2 HyperFlow: A Distributed Control Plane for OpenFlow

The initial design and implementation of SDN systems assumed a single controller for the sake of simplicity. However, as the number and size of production networks increases, relying on a single controller for the entire network might not be feasible for several reasons. First, the amount of control traffic destined towards the centralized controller grows with the number of switches. Second, if the network has a large diameter, no matter where the controller is placed, some switches will encounter long flow setup latencies. Finally, since the system is bounded by the processing power of the controller, flow setup times can grow significantly as demand grows with the size of the network. Figure 2.7a illustrates these issues in an OpenFlow-based network.

![Figure 2.7: A multi-site OpenFlow network with single and multiple controllers. Switch and controller association is depicted using colors and shadow pattern. a Deploying a single controller increases the flow setup time for flows initiated in site 2 and site 3 by 50ms. Also, an increase in flow initiation rates in the remote sites may congest the cross-site links. b In HyperFlow, all the requests are served by local controllers, and the cross-site control traffic is minimal: controllers mostly get updated by their neighbors.](image)

In this section, we present the design and implementation of HyperFlow, a distributed event-based control plane for OpenFlow, which allows operators deploy any number of controllers in their networks. HyperFlow provides scalability while keeping network con-
control logically centralized: all the controllers share the same consistent network-wide view and locally serve requests without actively contacting any remote node, thus minimizing the flow setup times. Additionally, HyperFlow does not require any changes to the OpenFlow standard [62] and only needs minor modifications to existing control applications. HyperFlow guarantees loop-free forwarding, and is resilient to network partitioning as well as component failures. Besides, it enables addition of administrative areas to OpenFlow to interconnect independently managed OpenFlow areas. Figure 2.7b shows how HyperFlow addresses the problems associated with a centralized controller in an OpenFlow network.

An alternative design is to keep the controller state in a distributed data store (e.g., a DHT) and enable local caching on individual controllers. Even though a decision (e.g., flow path setup) can be made for many flows by just consulting the local cache, inevitably some flows require state retrieval from remote controllers, resulting in a spike in the control plane service time. Additionally, this design requires modifications to applications to store state in the distributed data store. In contrast, HyperFlow proactively pushes state to other controllers, thereby enabling individual controllers to locally serve all flows. Also, HyperFlow’s operation is transparent to the control applications.

To the best of our knowledge, HyperFlow was the first distributed control plane for OpenFlow. More recent work [34, 51, 7, 92, 5, 53, 45] have followed this line of research and improved it different ways (see Section 2.3).

We implemented HyperFlow as an application for NOX [32]. The HyperFlow application is in charge of synchronizing controllers’ network-wide views (by propagating selected locally generated controller events), redirecting OpenFlow commands targeted to a non-directly-controlled switch to its respective controller, and redirecting replies from switches to the request-originator controllers. To facilitate cross-controller communications, we use publish/subscribe messaging paradigm. The HyperFlow’s design and implementation are discussed in the next section. Section 2.2.2 discusses when and why
a controller application must be modified to become HyperFlow-compatible. To evaluate HyperFlow, Section 2.2.3 estimates the maximum level of dynamicity a HyperFlow-based network can have while keeping the window of inconsistency among controllers bounded by a factor of delay between farthest controllers in the network.

2.2.1 Design and Implementation

A HyperFlow-based network is composed of OpenFlow switches as forwarding elements, NOX controllers as decision elements each running an instance of the HyperFlow controller application, and an event propagation system for cross-controller communication. All the controllers have a consistent network-wide view and run as if they are controlling the whole network. They all run the exact same controller software and set of applications. Each switch is connected to the best controller in its proximity. Upon controller failure, affected switches must be reconfigured to connect to an active nearby controller.\(^\text{10}\)

Each controller directly manages the switches connected to it and indirectly programs or queries the rest (through communication with other controllers). Figure 2.8 illustrates the high-level view of the system.

To provide an eventually consistent network-wide view among controllers, the HyperFlow controller application instance in each controller selectively publishes the events that change the global network view through a publish/subscribe system. Other controllers replay all the published events to reconstruct the state. This design choice is based on the following observations: (a) Any change to the network-wide view of controllers stems from the occurrence of a network event. A single event may affect the state of several applications, so the control traffic required for direct state synchronization grows with the number of applications, but it is bounded to a small number of events in our solution. (b) Only a very small fraction of network events cause changes to the network-wide view (on the order of tens of events per second for networks of thousands of hosts [32]). The

\(^\text{10}\)Currently, in our test environment, we used proprietary hardware vendor configuration interface to reconfigure the controller address.
Figure 2.8: High-level overview of HyperFlow. Each controller runs NOX with the HyperFlow application atop, subscribes to the control, data, and its own channel in the publish/subscribe system (depicted with a cloud). Events are published to the data channel and periodic controller advertisements are sent to the control channel. Controllers directly publish the commands targeted to a controller to its channel. Replies to the commands are published in the source controller.

The majority of network events (i.e., packet_in events) only request service (e.g., routing). (c) The temporal ordering of events, except those targeting the same switch, does not affect the network-wide view. (d) The applications only need to be minimally modified to dynamically identify the events which affect their state (unlike direct state synchronization which requires each application to directly implement state synchronization and conflict resolution).

**Event Propagation**

To propagate controller events to others, HyperFlow uses publish/subscribe messaging paradigm. The publish/subscribe system that HyperFlow uses must provide persistent storage of published events (to provide guaranteed event delivery), keep the ordering of events published by the same controller, and be resilient against network partitioning (i.e., each partition must continue its operation independently and upon reconnection, partitions must synchronize). The publish/subscribe system should also minimize the cross-site\textsuperscript{11} traffic required to propagate events, i.e., controllers in a site should get most traffic.

\textsuperscript{11}A site is a highly-connected component of the network with a large bisection bandwidth. However, the bandwidth and connectivity between regions is limited. Typically network devices in a single site...
of the updates of other sites from nearby controllers to avoid congesting the cross-region links. Finally, the system should enforce access control to ensure authorized access.

We implemented a distributed publish/subscribe system satisfying the above requirements using WheelFS [78]. WheelFS is a distributed file system designed to offer flexible wide-area storage for distributed applications. It gives the applications control over consistency, durability, and data placement according to their requirements via semantic cues. These cues can be directly embedded in the pathnames to change the behavior of the file system. In WheelFS, we represent channels with directories and messages with files. To implement notification upon message arrival (i.e., new files in the watched directories) HyperFlow controller application periodically polls the watched directories to detect changes.

Each controller subscribes to three channels in the network: the data channel, the control channel, and its own channel. All the controllers in a network are granted permissions to publish to all channels and subscribe to the three channels mentioned. The HyperFlow application publishes selected local network and application events which are of general interest to the data channel. Events and OpenFlow commands targeted to a specific controller are published in the respective controller’s channel. Additionally, each controller must periodically advertise itself in the control channel to facilitate controller discovery and failure detection. Access control for these channels are enforced by the publish/subscribe system.

HyperFlow is resilient to network partitioning because WheelFS is. Once a network is partitioned, WheelFS on each partition continues to operate independently. Controllers on each partition no longer receive the advertisements for the controllers on the other partitions and assume they have failed. Upon reconnection of partitions, the WheelFS nodes in both partitions resynchronize. Consequently, the controllers get notified of all the events occurred in the other partition while they were disconnected, and the network-
wide view of all the controllers converges.\textsuperscript{12}

Finally, we note that WheelFS can be replaced by any publish/subscribe system satisfying the above mentioned requirements. We chose WheelFS, primarily because not only it satisfies HyperFlow's requirements, but also enables us to rapidly build a prototype. However, as we show in Section 2.2.3, there is room for significant improvements to the existing publish/subscribe system.

\textbf{HyperFlow Controller Application}

HyperFlow application is a C++ NOX application we developed to ensure all the controllers have a consistent network-wide view. Each controller runs an instance of the HyperFlow application. Our implementation requires minor changes to the core controller code, mainly, to provide appropriate hooks to intercept commands and serialize events. Below, we describe the functions of the HyperFlow controller application.

\textbf{Initialization:} Upon NOX startup, the HyperFlow application starts the WheelFS client and storage services, subscribes to the network’s data and control channels, and starts to periodically advertise itself in the control channel. The advertisement interval must be larger than the highest round-trip time among controllers in a network. The advertisement message contains information about the controller including the identifiers of the switches it directly controls.

\textbf{Publishing events:} The HyperFlow application captures all the NOX built-in events (OpenFlow message events) as well as the events that applications register with HyperFlow. Then, it selectively serializes (using the Boost serialization library) and publishes the ones which are \textit{locally} generated and affect the controller state. For that, applications must be instrumented to tag the events which affect their state. Furthermore, applications should identify the parent event of any non-built-in event they fire. This way, HyperFlow can trace each high-level event back to the underlying lower-level event.

\textsuperscript{12}We note that this requires the network operator define a replication policy appropriate for the network setup.
Table 2.3: HyperFlow message naming convention. All message types contain the publisher controller id \((ctrl_id)\). Events and commands also contain an event identifier \((event_id)\) locally generated by the publisher. Commands also contain the identifier of the switch to which the command is targeted.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Name Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>(e: ctrl_id : event_id)</td>
</tr>
<tr>
<td>Command</td>
<td>(c: ctrl_id : switch_id : event_id)</td>
</tr>
<tr>
<td>Advertisement</td>
<td>(ctrl_id)</td>
</tr>
</tbody>
</table>

and propagate it instead. Using this method we ensure that the number of events propagated is bounded by the number of the OpenFlow message events generated by the local controller.

The name of the published messages contains the source controller identifier and an event identifier local to the publisher (see Table 2.3. This scheme effectively partitions the message namespace among controllers and avoids the possibility of any write conflicts. Moreover, a cached copy of a message (file) in our system never becomes stale. Therefore, using semantic cues, we instruct WheelFS to relax consistency requirements and fetch cached copies of files from neighboring controllers as fast as possible.

**Replaying events:** The HyperFlow application replays all the published events, because source controllers – with the aid of applications – selectively filter out and only publish the events necessary to reconstruct the application state on other controllers. Upon receiving a new message on the network data channel or the controller’s own channel, the HyperFlow application deserializes and fires it.

**Redirecting commands targeted to a non-local switch:** A controller can only program the switches connected directly to it. To program a switch not under direct control of the controller, the HyperFlow application intercepts when an OpenFlow message is about to be sent to such switches and publishes the command to the network control channel. The name of the published message shows that it is a command and also contains the source controller identifier, the target switch identifier, and the local command identifier (similar to the event message identifier).
Proxying OpenFlow messages and replies: The HyperFlow application picks up command messages targeted to a switch under its control (identified in the message name) and sends them to the target switch. To route the replies back to the source controller, the HyperFlow application keeps a mapping between the message transaction identifiers ($xid$) and the source controller identifiers. The HyperFlow application examines the $xid$ OpenFlow message events locally generated by the controller. If the $xid$ of an event is found in the $xid$-controller map, the event is stopped from being further processed and is published to the network data channel. The name of the message contains both controller identifiers. The original source controller picks up and replays the event upon receipt.

Health checking: The HyperFlow application listens for the controller advertisements in the network control channel. If a controller does not re-advertise itself for three advertisement intervals, it is assumed to have failed. The HyperFlow application fires a switch leave event for every switch that was connected to the failed controller. Upon controller failure, HyperFlow configures the switches associated with the failed controller to connect to another controller. Alternatively, either nearby controllers can serve as a hot standby for each other to take over the IP address.

### 2.2.2 Requirements on Controller Applications

For most controller applications, HyperFlow only requires minor modifications: they must dynamically tag events which affect their state. However, some of them must be further modified to ensure correct operation under temporal event reordering and transiently conflicting controller views, and guarantee scalability. Besides, we discuss how the controller applications must be modified to enable interconnection of independently-managed OpenFlow networks.

Event reordering: In HyperFlow, correct operation of control applications must not depend on temporal ordering of events except those targeting the same entity (e.g., the same switch or link), because different controllers perceive events in different orders.
Besides, resilience to network partitioning requires control applications to tolerate out-of-order event delivery (even lagging several hours) without sacrificing correctness, because each partition is notified of the state of the other partitions upon reconnection.

**Correctness:** Transient inconsistencies among controllers may lead to conflicting decisions. To ensure correct operation in all cases, control applications must forward requests to the *authoritative* controller. The authoritative controller for a given flow is the one managing the flow’s source switch. Consider the switching/routing applications as an example: To ensure loop-free forwarding, flow paths must be set up by the controller managing the flow’s source switch. Other controllers must redirect the request to the authoritative controller in case they receive a flow initiation event. As another example, consider a network with a policy which requires both the forward and reverse paths of all flows to match. To guarantee this, the source controller must simultaneously set up both paths upon flow initiation. This modification ensures that the policy is always correctly enforced from the source controller’s perspective.

**Bounded number of possibly effective events:** The number of events which possibly affect the state of a HyperFlow-compliant application must be bounded by $O(h + l + s)$, where $h$ is the number of hosts, $l$ is the number of links, and $s$ is the number of switches in the network. In other words, applications whose state may be affected by $O(f(n))$ events, where $f(n)$ is any function of the number of flows in the network, incur a prohibitively large overhead and must be modified.

**Measurement applications:** Applications which actively query the switches perform poorly under HyperFlow, because the number of queries grows linearly with the number of controllers. Such applications must be modified to partition queries among controllers in a distributed fashion and exchange the results (encapsulated in self-defined events) using HyperFlow. Consider the discovery application as an example. The discovery application must only sends link layer discovery protocol (LLDP) probes out of the switches under its direct control. For each link between a directly and a non-directly controlled switch
pair, the discovery application receives an LLDP packet generated by another controller signaling the connectivity. Finally, the discovery application must propagate its link events using HyperFlow.

**Interconnecting HyperFlow-based OpenFlow networks:** To interconnect two independently managed HyperFlow-based OpenFlow networks (areas), controller applications need to be modified to be made area-aware. They must listen for area discovery events from HyperFlow, enforce the area policies declared using a policy language (e.g., Flow-based Management Language [36]), and exchange updates with the neighboring area through a secure channel providing publish/subscribe service. Applications should encapsulate updates in self-defined events, and have HyperFlow propagate them to the neighboring areas. HyperFlow removes the need for individual control applications to discover their neighbors and communicate directly; instead, control applications just fire events locally and HyperFlow delivers them to neighbors. We note that the implementation of this part is not completed yet.

### 2.2.3 Evaluation

We performed a preliminary evaluation to estimate the maximum level of network dynamicity it can support while guaranteeing a bounded inconsistency window among controllers. Throughout our experiments we used ten servers each equipped with a gigabit NIC and running as a WheelFS client and storage node. In the near future, we plan to deploy HyperFlow on a large testbed and characterize its performance, robustness, and scalability with a realistic network topology and traffic.

Each NOX instance can handle about 30k flow installs per second [82]. However, it can typically process a far larger number of events which do not trigger an interaction with a switch. Events published through HyperFlow only affect controller state and should not trigger any interaction with controllers. Therefore, using HyperFlow, network operators can easily add more controllers to handle more flow initiation events while keeping the flow
setup latency minimal. We note that, since in HyperFlow controllers’ operations do not depend on other controllers, they continue to operate even under heavy synchronization load. However, as the load increases, the window of inconsistency among controllers grows (i.e., the time it takes to have the views converge).

To find the number of events that HyperFlow can handle while providing a bounded inconsistency window among controllers, we benchmarked WheelFS independently to find the number of 3-KB sized files (sample serialized $\text{datapath} \_ \text{join}$ event using the XML archive$^{13}$) we can write (publish) and read. For that, we instrumented the HyperFlow application code to measure the time needed to read and deserialize (with eventual consistency), as well as serialize and write (write locally and don’t wait for synchronization with replicas) 1000 such files. We ran each test 10 times and averaged the results. HyperFlow can read and deserialize 987, and serialize and write 233 such events in each second. The limiting factor in this case is the number of reads, because multiple controllers can publish (write) concurrently.

Based on the above analysis, assuming adequate control bandwidth, HyperFlow can guarantee a bounded window of inconsistency among controllers, if the network changes trigger less than around 1000 events per second (i.e., total of 1000 switch and host joins and leaves, and link state changes). We may be able to improve HyperFlow’s performance by modifying WheelFS (whose implementation is not mature yet) or designing an alternative publish/subscribe system. Finally, we note that HyperFlow can gracefully handle spikes in network synchronization load without losing any events, however in that period the controller views converge with an added delay.

$^{13}$This is significantly larger than most serialized events. Also, in real deployments we should use binary archives which significantly reduces message sizes.
2.3 Concluding Remarks

Since we published the controller performance study [88] and HyperFlow [87], a rich body of literature has followed this line of research and improved it in different ways (see [34, 51, 7, 92, 5, 53, 45] for a small sampling). HyperFlow (similar to the legacy control planes) settles for eventual consistency. Even though this is sufficient for some control state (e.g., link state), stronger consistency may be required for others (e.g., consistent network-wide resource allocation and accounting). Onix [51], OpenDaylight [61], and ONOS [7] provide mechanisms (e.g., using ZooKeeper [37]) to deliver stronger consistency guarantees and distributed state management. Kandoo’s [34] hierarchical design (in contrast to HyperFlow’s flat design) differentiates between local and global control applications where local ones serve requests with their local state (as opposed to the network-wide state). Ravana [45] goes beyond control state consistency and ensures datapath state is handled consistently after a failure.

Finally, Maple [92] and Beacon [5] have demonstrated significantly improved performance over NOX-MT.

A performant and scalable control plane alone is required but not sufficient to have a performant network. With the canonical SDN (fixed function datapath), it is possible to implement a policy that has a much larger memory footprint than the aggregate flow table capacity in datapath or even a policy that requires functions unavailable in datapath. All it takes is to rely on the generality of the SDN control platform to fill this gap. In this setting, the control platform ends up reactively processing possibly a large volume traffic. Even though this possibility is enticing, we note that it would significantly degrade the network performance. In the next chapter, we present an alternative SDN design that introduces flexibility and modularity in datapath. Embracing this design takes us a step closer to systematically guarantee performance objectives in SDN.
Chapter 3

Evolving the Canonical SDN

The canonical SDN paradigm is built around logically centralized controllers and the OpenFlow dataplane specification. OpenFlow was designed with the assumption that switches are based on ASICs with limited forwarding flexibility; in contrast, the design of the control plane assumes controllers are fully programmable. This dichotomy between a constricted dataplane and general control plane leads to several problems, many of which have been admirably addressed in the recent SDN literature. Some works (e.g., [95, 19]) push the state to the datapath as much as possible and as such are constrained to policies that could be efficiently implemented with the in-datapath memory. Others (e.g., [56]) provide high-level language abstractions for describing network policy and compile it down to rules installed on hardware switches and serve packets the state for which could not be pushed to hardware in the control plane. This approach provides the flexibility by depending on the generality of SDN control plane. But it remains challenging to provide both flexibility and performance.

Instead, to provide guaranteed performance levels, this thesis considers an alternative SDN design which distinguishes between the functions of network edge and core and provides a far more flexible datapath by embracing software packet processing at the network edge. Neither of these are foreign concepts to networks. Networks today are
replete with middleboxes [74] that provide flexibility in datapath, and more recently, Network Function Virtualization (NFV), which has to this point been largely promoted on the basis of easier deployability, uses software switching at the network edge. MPLS distinguishes between the network edge and core with core providing simple packet delivery. Our contribution here is to combine these into a coherent architecture.

The modularity we introduce in the datapath function enables us to deconstruct SLA enforcement into independent pieces. To meet an SLA objective, the core should be provisioned to provide the necessary packet delivery guarantees. The other functions and their corresponding control planes could then be independently provisioned and, in case they are not co-located, interconnected using the fabric. This thesis looks at performance isolation for colocated software datapaths (Chapter 4) as well as performance and scalability of the control platform (Chapter 2), but assumes a QoS-enabled fabric.

In this chapter\footnote{Part of the work presented in this chapter first appeared in \cite{15} \copyright 2012 Association for Computing Machinery, Inc. \url{http://doi.acm.org/10.1145/2342441.2342459}. Reprinted by permission.}, we first study some limitations of the canonical SDN (Section 3.1). In Section 3.2, we present a high-level overview of the architecture. The interfaces and functions of the fabric are presented in Section 3.3. Section 3.4 looks at the implications of software edge. We discuss the reasons this architecture will serve us better in the long term in Section 3.5.

## 3.1 SDN’s Shortcomings

### 3.1.1 SDN’s Need for Fabric

Network infrastructure design is guided by network requirements and network interfaces. Network requirements come from two sources: hosts and operators. Hosts (or, more accurately, the users of that host) want their packets to travel to a particular destination, and they may also have QoS requirements about the nature of the service these packets
receive en route to that destination. Network operators have a broader set of requirements — such as traffic engineering, virtualization, tunneling and isolation — some of which are invisible and/or irrelevant to the hosts. As we observe below, the control mechanisms used to meet these two sources of requirements are quite different.

Like any system, networks can be thought of in terms of interfaces; here we use that term not to refer to a formal programmatic interface, but to mean more generally and informally places where control information must be passed between network entities. There are three relevant interfaces we consider here:

- **Host — Network**: The first interface is how the hosts inform the network of their requirements; this is typically done in the packet header (for convenience, in the following we will focus on L3, but our comments apply more generally), which contains a destination address and (at least theoretically) some ToS bits. However, in some designs (such as IntServ), there is a more explicit interface for specifying service requirements.

- **Operator — Network**: The second interface is how operators inform the network of their requirements; traditionally, this has been through per-box (and often manual) configuration, but SDN (as we discuss later) has introduced a programmatic interface.

- **Packet — Switch**: The third interface is how a packet identifies itself to a switch. To forward a packet, a router uses some fields from the packet header as an index to its forwarding table; the third interface is the nature of this index.

We now turn to how the original Internet, MPLS, and SDN deal with requirements and implement these interfaces.

In the original Internet design, there were no operator requirements; the goal of the network was to merely carry the packet from source to destination (for convenience, we will ignore the ToS bits), and routing algorithms computed the routing tables necessary
to achieve that goal. At each hop, the router would use the destination address as the key for a lookup in the routing table; that is, in our conceptual terms, every router would independently interpret the host requirements and take the appropriate forwarding action. Thus, the Host-Network and Packet-Switch interfaces were identical, and there was no need for the Operator-Network interface.

MPLS introduced an explicit distinction between the network edge and the network core. Edge routers inspect the incoming packet headers (which express the host’s requirements as to where to deliver the packet) and then attach a label onto the packet which is used for all forwarding within the core. The label-based forwarding tables in core routers are built not just to deliver packets to the destination, but also to address operator requirements such as VPNs (tunnels) or traffic engineering. MPLS labels have meaning only within the core, and are completely decoupled from the host protocol (e.g., IPv4 or IPv6) used by the host to express its requirement to the network. Thus, the interface for specifying host requirements is still IP, while the interface for packets to identify themselves is an MPLS label. However, MPLS did not formalize the interface by which operators specified their control requirements. Thus, MPLS distinguished between the Host-Network and Packet-Switch interfaces, but did not develop a general Operator-Network interface.

In contrast to MPLS, SDN focuses on the control plane. In particular, SDN provides a fully programmatic Operator-Network interface, which allows it to address a wide variety of operator requirements without changing any of the lower-level aspects of the network. SDN achieves this flexibility by decoupling the control plane from the topology of the data plane, so that the distribution model of the control plane need not mimic the distribution of the data plane. While the term SDN can apply to many network designs with decoupled control and data planes [73, 10, 31, 13, 14], we will frame this discussion around its canonical instantiation: OpenFlow.²

²We note that there has been a wealth of recent work on SDN, including [28, 67, 12, 58, 91, 75, 95, 5, 89], which extends SDN in one or more directions, but all of them are essentially orthogonal to the
In OpenFlow each switch within the network exports an interface that allows a remote controller to manage its forwarding state. This managed state is a set of forwarding tables that provide a mapping between packet header fields and actions to execute on matching packets. The set of fields that can be matched on is roughly equivalent to what forwarding ASICs can match on today, namely standard Ethernet, IP, and transport protocol fields; actions include the common packet operations of sending the packet to a port as well as modifying the protocol fields. While OpenFlow is a significant step towards making the control plane more flexible, it suffers from a fundamental problem that it does not distinguish between the Host-Network interface and the Packet-Switch interface. Much like the original Internet design, each switch must consider the host’s original packet header when making forwarding decisions. Granted, the flexibility of the SDN control plane allows the flow entries, when taken collectively, to implement sophisticated network services (such as isolation); however, each switch must still interpret the host header.\footnote{One could, of course, use SDN to implement MPLS. However, each switch must be prepared to deal with full host headers.}

This leads to three problems:

- First, it does not fulfill the promise of simplified hardware. In fairness, OpenFlow was intended to strike a balance between practicality (support matching on standard headers) and generality (match on all headers). However, this requires switch hardware to support lookups over hundreds of bits; in contrast, core forwarding with MPLS need only match over some tens of bits. Thus, with respect to the forwarding hardware alone, an OpenFlow switch is clearly far from the simplest design achievable.

- Second, it does not provide sufficient flexibility. We expect host requirements to continue to evolve, leading to increasing generality in the Host-Network interface, which in turn means increasing the generality in the matching allowed and the issues we are discussing here.
actions supported. In the current OpenFlow design paradigm, this additional generality must be present on every switch. It is inevitable that, in OpenFlow’s attempt to find a sweet spot in the practicality vs generality tradeoff, needing functionality to be present on every switch will bias the decision towards a more limited feature set, reducing OpenFlow’s generality.

- Third, it unnecessarily couples the host requirements to the network core behavior. This point is similar to but more general than the point above. If there is a change in the external network protocols (e.g., switching from IPv4 to IPv6) which necessitates a change in the matching behavior (because the matching must be done over different fields), this requires a change in the packet matching even in the network core.

Thus, our goal is to extend the SDN model in a way that avoids these limitations yet still retains SDN’s great control plane flexibility. To this end, it must retain its programmatic control plane interface (so that it provides a general Operator-Network interface), while cleanly distinguishing between the Host-Network and Packet-Switch interfaces (as is done in MPLS).

### 3.1.2 Challenges of ASIC-driven Networking

Combining the several goals described earlier, the canonical SDN paradigm revolves around a dataplane built out of ASIC-based switches that implement the OpenFlow specification and a control plane based on x86-based controllers that support a network operating system and applications built thereon. This leads to a stark dichotomy in capabilities: the dataplane has limited header matching and a fixed set of forwarding actions, while the control plane is fully programmable.

This dichotomy has led to a disconnect between commercial and academic investigations; most early academic interest on SDN focused on reactive designs (starting with
Ethane [13]) where the majority of forwarding decisions are made first in the controller and then cached in switches, while most commercial designs focus more on proactive approaches where almost all forwarding decisions are pre-emptively pushed to switches. This split in focus is natural, since the reactive approach is more flexible (and hence more attractive to researchers) while the proactive approach is more scalable and predictable (and hence more deployable in commercial settings). Of course, real implementations often combine aspects of both approaches, but the difference in emphasis between academic and commercial designs is striking.

In addition, this stark difference in capabilities confronts system builders with several thorny problems, including (as we discuss below) how to coordinate forwarding between the control and data planes, how to deal with severe resource limitations on the forwarding plane, and how to compose different functionalities. There has been a significant line of research devoted to solving these problems [28, 56, 57, 91]; this research is ingenious and has in particular led to a much deeper understanding of how domain-specific programming languages can help in SDN design.

Despite the benefits of centralization, developing control logic for SDN controllers is both challenging and error prone. In this section we review some common problems faced by SDN developers and discuss proposals from the community for solving these issues. These solutions leverage a wide variety of prior research and manifest as a variety of tools including new domain specific programming languages, development frameworks, extensions to existing languages, etc. We also show that the exclusive use of ASICs and the corresponding limits of the forwarding model underlie these problems.

**Deciding Which Packets Controller Should Process**

In the canonical SDN model packets which need complex processing that cannot be carried out at the switch are offloaded to the controller. A packet is however exclusively processed at either the switch or the controller and programmers must program the
switch (using an appropriate set of processing rules) so that the correct subset of packets is sent to the controller. Application commonly require only a fraction of the packets to undergo special processing at the controller. For instance, an authentication application might require special processing for only the first packet of a transport connection, and a logging application might only require complex processing for every fifth packet. The latency incurred in bringing a packet to the control plane and processing at the controller is much larger than at a switch [49]. Developers are therefore required to carefully craft the set of rules installed at the switch so that the controller receives a sufficient set of packet (thus ensuring correctness), while maximizing the number of packets handled in the switch (thus achieving efficiency).

Instead of requiring programmers to craft rules using low level primitives provided by OpenFlow, Frenetic [28] provides developers with a domain specific language and framework to declaratively specify the set of packets that need to be processed by their application. Using these declarations, Frenetic then determines the set of packets to be processed at the controller, and constructs an appropriate set of rules that can be installed at the switch to process the rest. Furthermore to guide developers in constructing these declarative rules and reason about the performance Frenetic also provides a query cost model associated with these declarative rules.

A separate concern is the possibility of race conditions when installing new rules in a switch. SDN applications, in response to packets sent to the controller, commonly generate and install a set of rules at switches in the data path. These rules cache the results of processing carried out at the controller and are used to process and forward subsequent packets in the same flow (or packets that are otherwise similar). However, more packets belonging to the flow might arrive at the controller while it is computing and installing these rules, and these packets might trigger unnecessary recomputations or result in conflicting flow entries being installed. Since the switch is not aware of the granularity at which the controller processes packets, there’s little it can do to prevent
the race conditions. Several controller frameworks, including Frenetic, resolve this issue by caching rules in the controller before installing them on the switch. Semantically applications then receive a single packet at a time, and generated rules apply to all subsequent packets. This is semantically identical to implementing all packet processing required by an application in the data path.

**Lookup Resource Limitations**

Controllers which restrict themselves to features available in the ASIC might still need to implement part of their functionality in the controller to overcome the lookup resource limitations in the switch. To overcome the limitations for lookup table sizes, developers use the switch as a cache for computed rules. To minimize the impact for the throughput, maximizing the cache hit rate *i.e.* maximizing the number of packets that match the set of rules cached in the switch is a priority. While using wildcarded OpenFlow flow entries to match a wider range of packets (effectively increasing the amount of cache available) calculating these wildcard entries can be quite complicated, especially since the developer must often translate a set of predicates combined using boolean operators to a single wildcard expression, and do this while considering already installed flow entries.

NetCore [56] provides a compiler and tools which can generalize flow entries generated at the controller. When using NetCore, unmatched packets are sent to the controller. The control application specifies the set of predicates used to determine the action applicable to the packet, and NetCore can then translate these predicates to a new wildcard flow entry without violating the overall correctness of the system.

**Composing Packet Operations**

It is convenient to express complex packet processing as a series of isolated modules. The packets may be processed in a specific order (referred to as sequential composition in [57]), or be independently as on copies of the same packet (referred to as parallel
composition). While all parallel compositions can be expressed as one or more serial composition pipelines\(^4\), doing so might result in performance degradation, especially when processing is offloaded to the controller (since in this case packet forwarding might be needlessly delayed while waiting for a module to finish processing a packet).

Canonical SDN model provides no support for sending a copy of a packet to the controller, and instead a packet has to be processed either entirely at the controller, or entirely at the switch. That is, a packet can only be processed sequentially and not by both controller and switch in parallel. This implies that some sequential arrangements perform significantly better than others. Instead of requiring the developer to determine the best arrangement for modules, frameworks such as Frenetic and Pyretic allow developers to specify whether modules should be composed sequentially or in parallel, and then let the framework to take care of selecting the optimal composition. For parallel composition, Frenetic uses semantic information about each of the modules (including information provided by the declarations described previously) and compute a sequential arrangement of modules to minimizes the set of packets sent to the controller.

For sequential composition, Pyretic offers a developer with a framework for implementing individual modules, and tools to combine and split modules implemented using the framework. This shields the developer from the low-level details of determining what rules are installed on particular switches, and the set of actions that the controller must perform. For instance, in cases where the number of modules exceeds the number of switches available in a path, developer would have to compute rules that somehow combine the actions performed by several modules. Similarly in cases where the number of switches exceeds the number of modules the developer would have to split a single module across several switches.

All of the previously stated problems can be traced back to limitations (both in functionality and resources) of the forwarding model, and an attempt to fit a rich set of

\(^{4}\text{This is similar to how a set of parallel processes can be emulated on a single sequential processor.}\)
Chapter 3. Evolving the Canonical SDN

3.2 Overview

The discussion in the previous section shows that many common problems faced by SDN developers are a result of an inflexible and limited forwarding model. The current forwarding model is derived from features commonly available in commodity switches. While this model is updated (for instance with new OpenFlow version) with features added by hardware switches, these updates tend to be minor and are not necessarily informed by applications.

We believe that SDN must move away from the current paradigm which centers on universal support for OpenFlow and heavy reliance on hardware switches. Our proposal is quite different, in that it obviates the need for OpenFlow and relies on software packet processing for all complex functionality. In this subsection, we lay out the essentials of our proposal with its two key architecture principles.

Distinguishing between edge and core

Our design’s first architectural principle is that there is a fundamental distinction between the network edge and the network core. By this we mean that the network core is implemented as a “fabric” (in the parlance that is common in datacenters) that supports basic edge-to-edge connectivity (unicast, and often also multicast and anycast), along with QoS (e.g., priority levels) and group management.

Fabrics can be implemented in many ways. To minimize deployment barriers, one can implement a fabric using standard IP forwarding, with traditional routing protocols providing the edge-to-edge connectivity and group membership. However, if one is building this in a clean-slate fashion, the minimal forwarding capability required is MPLS-like label-switching (with priority forwarding), with group membership handled by an SDN
platform dedicated to the fabric. Since the fabric interface is all that matters to the rest of the design, and there are a variety of acceptable design choices in building such fabrics, we only discuss its high-level components in Section 3.3.

The edge handles everything else besides basic edge-to-edge connectivity; most importantly, this includes processing packet headers and deciding to which edge port a packet should be sent. Thus, only the edge needs to understand the IP protocol while the fabric can remain oblivious to the nature of host protocols. We use an SDN platform to control these edge nodes, running on a set of associated SDN controllers (the “edge controllers”).

This clean division of labor between edge and core is clear for many tasks (routing, isolation, access control, etc.), but becomes more complicated when we consider traffic engineering (or load balancing as it is referred to in datacenters and enterprises). Traffic engineering (TE) can be implemented within this edge/core framework in a variety of ways. One could use standard TE approaches (MPLS rerouting, OSPF weight adjustment, or more recent mechanisms as in [4]) inside the fabric, so that TE is nothing more than an internal mechanism used to improve the quality of the service delivered by the fabric. Alternatively, the fabric interface could be extended to support multiple paths, and one could adopt an edge-based approach such as described in [43, 42, 35] to balance load across these paths. We could use either approach, but for convenience, in this thesis, we assume that TE is handled internally by the fabric.

**Software processing at the edge**

The second architectural principle is the availability of software for performing packet forwarding and processing at the edge.\(^5\) We should clarify that we are not opposed to various forms of hardware-assist architectures (*e.g.* enabling some of the simpler

---

\(^5\)Note that we artificially restrict ourselves to the extreme cases of fully custom ASICs and fully commodity processors. There is a point in between, namely network processors, that may be applicable in some circumstances. However, the market (particularly the NFV effort) is largely focused on the commodity approach, so we adopt that here.
forwarding tasks to be done in hardware) but (i) we do not think it necessary in general, though it might be useful for tasks such as encryption, and (ii) our goal is to make software ubiquitously available at the edge, not to prevent hardware forwarding if that is desired. The key point is that in this design, hardware does not limit what can be done at the edge, it merely assists when it can.

The obvious advantage of this approach is flexibility. Rather than having some ASIC-determined set of forwarding primitives, the network edge can now support any form of packet processing desired. This enables the easy deployment of software middleboxes as in NFV, as well as broadens the scope of basic forwarding actions.

This is certainly not the first time these architectural points have been made, but they were previously made in a piecemeal fashion in a series of workshop papers (see, for example, [15, 64]) directed towards datacenters. In this thesis, we have integrated these observations into a coherent architecture and explored its long-term implications. Moreover, the academic literature on SDN has focused almost exclusively on the canonical paradigm (of hardware switches supporting OpenFlow); as we discuss in Section 3.4, many of the problems being addressed in the current SDN literature would be rendered moot by our design.

3.3 Fabric

In this section we explore how the SDN architectural framework might be extended to better meet the goals listed in the introduction. Our proposal is centered on the introduction of a new conceptual component which we call the “network fabric”. While a common term, for our purposes we limit the definition to refer to a collection of forwarding elements whose primary purpose is packet transport. Under this definition, a network fabric does not provide more complex network services such as filtering or isolation.

The network then has three kinds of components (see Figure 3.1): hosts, which act
as sources and destinations of packets; edge switches, which serve as both ingress and egress elements; and the core fabric. The fabric and the edge are controlled by (logically) separate controllers, with the edge responsible for complex network services while the fabric only provides basic packet transport. The edge controller handles the Operator-Network interface; the ingress edge switch, along with its controller, handle the Host-Network interface; and the switches in the fabric are where the Packet-Switch interface is exercised.

The idea of designing a network around a fabric is well understood within the community. In particular, there are many examples of limiting the intelligence to the network edge and keeping the core simple.\(^6\) Thus, our goal is not to claim that a network fabric is a new concept but rather we believe it should be included as an architectural building block within SDN. We now identify the key properties for these fabrics.

**Separation of Forwarding.** In order for a fabric to remain decoupled from the edge it should provide a minimal set of forwarding primitives without exposing any internal forwarding mechanisms that would be visible from the end system if the fabric were replaced. We describe this in more detail below but we believe it is particularly important that external addresses are not used in forwarding decisions within the fabric both to simplify the fabric forwarding elements, but also to allow for independent evolution of fabric and edge.

**Separation of Control.** While there are multiple reasons to keep the fabric and the edge’s control planes separate, the one we would like to focus on is that they are solving two different problems. The fabric is responsible for packet transport across the network, while the edge is responsible for providing more semantically rich services such as network

---

\(^6\)This is commonly done for example in datacenters where connectivity is provided by a CLOS topology running an IGP and ECMP. It is also reflected in WANs where interdomain policies are implemented at the provider edge feeding packets into a simpler MPLS core providing connectivity across the operator network.
security, isolation, and mobility. Separating the control planes allows them each to evolve separately, focusing on the specifics of the problem. Indeed, a good fabric should be able to support any number of intelligent edges (even concurrently) and vice versa.

Note that fabrics offer some of the same benefits as SDN. In particular, if the fabric interfaces are clearly defined and standardized, then fabrics offer vendor independence, and (as we describe in more detail later) limiting the function of the fabric to forwarding enables simpler switch implementations.

3.3.1 Fabric Service Model

Under our proposed model, a fabric is a system component which roughly represents raw forwarding capacity. In theory, a fabric should be able to support any number of edge designs including different addressing schemes and policy models. The reverse should also be true; that is, a given edge design should be able to take advantage of any fabric regardless of how it was implemented internally.

The design of a modern router/switch chassis is a reasonably good analogy for an SDN architecture that includes a fabric. In a chassis, the line cards contain most of the
Table 3.1: Fabric service model. Note that the ToS bits in the packet for QoS within the fabric are not included above.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attach(P)</td>
<td>Attach a fabric port P to the fabric.</td>
</tr>
<tr>
<td>Send(P, pkt)</td>
<td>Send a packet to a single fabric port.</td>
</tr>
<tr>
<td>Send(G, pkt)</td>
<td>Send a packet to a multicast group G.</td>
</tr>
<tr>
<td>Join(G, P)</td>
<td>Attach a port to a multicast group G.</td>
</tr>
<tr>
<td>Leave(G, P)</td>
<td>De-attach a port from a multicast group G.</td>
</tr>
</tbody>
</table>

intelligence and they are interconnected by a relatively dumb, but very high bandwidth, backplane. Likewise, in an SDN architecture with a fabric, the edge will implement the network policy and manage end-host addressing, while the fabric will effectively interconnect the edge as fast and cheaply as possible.

The chassis backplane therefore provides a reasonable starting point for a fabric service model. Generally, a backplane supports point-to-point communication, point-to-multipoint communication, and priorities to make intelligent drop decisions under contention. In our experience, this minimal set is sufficient for most common deployment scenarios. More complex network functions, such as filtering, isolation, stateful flow tracking, or port spanning can be implemented at the edge. Table 3.1 summarizes this high-level service model offered by the fabric.

### 3.3.2 Fabric Path Setup

Another consideration is path setup. In the “wild” two methods are commonly used today. In the datacenter, a common approach is to use a standard IGP (like OSPF) and ECMP to build a fabric. In this case, all paths are calculated and stored in the fabric. MPLS, on the other hand, requires the explicit provisioning of an LSP by the provider. The primary difference between the two is that when all forwarding state is precalculated, it is normally done with the assumption that any point at the edge of the fabric can talk to any other point. On the other hand, provisioned paths provide an isolated forwarding context between end points that is dictated by network operator
(generally from the provider edge).

We believe that either model works in practice depending on the deployment environment. If both the edge and the fabric are part of the same administrative domain, then precalculating all routes saves operational overhead. However, if the edge and fabric have a customer-provider relationship, then an explicit provisioning step may be warranted.

### 3.3.3 Addressing and Forwarding in the Fabric

As we have described it, a forwarding element in the fabric differs from traditional network forwarding elements in two ways. First, they are not required to use end-host addresses for forwarding, and second, they are only responsible for delivering a packet to its destination(s), and not enforcing any policy. As a result, the implementation of the fabric forwarding element can be optimized around relatively narrow requirements. Two current approaches exemplify the options available:

- One option would be to follow MPLS and limit network addresses to opaque labels and the forwarding actions to forward, push, pop and swap. This would provide a very general fabric that could be used by multiple control planes to provide either path-based provisioning or destination-based forwarding with label-aggregation.

- Another option would be to limit the packet operations to a destination address lookup with a longest prefix match with ECMP-based forwarding. It is unlikely that this would be suitable for path-based provisioning, but it would likely result in a simpler control plane and higher port densities.

Our preference is to use labels similar to MPLS because it supports a more general forwarding model. However, our main point here is that the SDN architecture should incorporate the notion of a fabric, but SDN need not be concerned with the specifics of the fabric forwarding model (indeed, that is the point of having a fabric!). Because
the fabric can evolve independently of the edge, multiple forwarding models can exist simultaneously.

### 3.3.4 Mapping the Edge Context to the Fabric

The complexity in an edge/fabric architecture lies in mapping the edge context to network addresses or paths. By “mapping” we simply mean figuring out which network address or label to use for a given packet. That is, when a packet crosses from the edge to the fabric, something in the network must decide with which fabric-internal network address to associate with the packet. There are two primary mechanisms for this:

**Address translation.** Address translation provides the mapping by swapping out addresses in situ. For example, when the packet crosses from the edge to the network, the edge addresses are replaced with fabric internal addresses, and then these addresses are translated back into appropriate edge addresses at the destination. The downside of this approach is that it unnecessarily couples the edge and network addressing schemes (since they would need to be of the same size, and map one-to-one with each other).

**Encapsulation.** A far more popular, and we believe more general, approach to mapping an edge address space to the fabric-internal address space is encapsulation. With encapsulation, once a packet crosses from the edge to the network, it is encapsulated with another header that carries the network-level identifiers. On the receiving side, the outer header is removed.

In either case (address translation or encapsulation), a lookup table at the edge must map edge addresses to network addresses to get packets across the fabric. However, unlike the fabric forwarding problem, this lookup may include any of the headers fields that are used by the edge. This is because more sophisticated functions such as filtering, isolation, or policy routing (for example, based on the packet source) must be implemented at the
edge. There are many practical (but well understood) challenges in implementing such a mapping that we will not cover. These include the control plane (which must maintain the edge mappings), connectivity fault management across the fabric, and the impact of addressing to basic operations and management.

3.4 Implications of Software Edge

A software switch, by virtue of being implemented in software running on commodity hardware, offers the same packet processing functionality as a SDN controller, eliminating the need to offload packet processing to the controller. This eliminates the associated problems.

In this new model application developers must still choose where to place functionality. This decision should be made so as to minimize the amount of network traffic required by the application. A networking application (this includes applications in traditional networks, current SDN networks, etc.) can choose to either bring packets to the control plane where policy state reside or to push this policy state to the data plane where packets are normally processed. For a majority of applications the size of the policy state is small, while the number of packets that can be processed using this state is nearly unbounded, and hence placing the application in the data plane makes sense for most applications and we expect most developers to develop applications that are entirely implemented in the data plane.

The controller, and the control plane in such a model is then responsible merely for computing and managing this policy state and for coordinating this state across the entire edge. Some applications, which rely on frequently changing, closely coordinated state, for instance control applications such as a DHCP agent might still be implemented in the controller. These applications are generally required to process a very small number of packets and rely on state that changes almost as frequently as packets arrive.
Composition also becomes an easier problem in our model. Software packet processors are already often written as a pipeline consisting of several modules which act in concert [50, 64]. This is no different than a sequential pipeline of modules through which a packet is processed. Furthermore this pipelined architecture allows software switches to present as large or as small a number of tables as the number of components, thus eliminating the need to explicitly combine multiple tables into one. The work on distribution one or a few components onto several switches is however still applicable, and allows a single virtual switch to be constructed out of several edge switches.

Finally, while the use of a forwarding model with sufficient resources eliminates the caching problem faced by current SDN networks, the necessity for caching is not eliminated in our model. In current SDN networks application can either choose to proactively install rules on a switch, or reactively compute new rules in response to controller events. An application might chose to reactively install rules in cases where either a proactive computation would either yield a large set of rarely used rules or in cases where an applications decision depends on dynamic state (for instance on the current time, number of flows, etc.). This choice is orthogonal to where application functionality resides.

Reactive applications commonly rely on caching to reduce the average processing time per packet. Such applications precompute a set of forwarding rules based on a snapshot of dynamic state, and require that these rules be invalidated when such state changes. Cache invalidation can either be manually carried out by the application developer or be provided by the framework. Since tracking dependencies per rule is both expensive and error-prone we envision the use of automated tools for dependency tracking. In the PL community dataflow analysis [48] is often used to perform such analysis and relies on propagating dependencies for a computed value. In principle this is similar to the kind of analysis enabled by the use of declarative predicates in NetCore. However NetCore in its does not compute dependencies for cached rules and does not evict rules when policy state changes. Further work is therefore required to realize tools for effective
3.5 Discussion

Isn’t this just another approach to layering? To some extent, one could view the edge and the core as different layers, with the edge layer running “over” the core layer; to that extent this is indeed just another approach to layering. And layering provides some of the same benefits we claim: it decouples the protocols in different layers (thereby increasing innovation) and allows for different layers to have different scopes (which is important for scaling). However, current dataplane layering can be thought of as “vertical”, making distinctions based on how close to the hardware a protocol is, and each layer goes all the way to the host. When a layer is exposed to the host, it becomes part of the host-network interface.

What we are proposing here is more of a “horizontal” layering, where the host-network interface occurs only at the edge, and the general packet-switch interface exists only in the core. This is a very different decoupling than provided by traditional layering. We not only want to decouple one layer from another, we want to decouple various pieces of the infrastructure from the edge layer entirely.

Where is the edge? The edge of a datacenter network is not the same as the edge of a carrier or a campus network. When the network serves no purpose other than interconnecting endpoints, the edge extends to the network stack of individual endpoints whereas in a multi-tenant datacenter the hypervisor vSwitch may constitute its edge. A carrier network’s edge may run on a cluster of servers implementing numerous complex policies. An edge endpoint of a large network may be constructed recursively based on the SDN design presented in this chapter. For instance, in a campus network, the backbone network provides a fabric that interconnects different departments (edge). A departmental network provides a fabric to interconnect individual research groups’ facilities (edge) and
so on.

**What does this mean for OpenFlow?** This approach would require an “edge” version of OpenFlow, which is much more general than today’s OpenFlow, and a “core” version of OpenFlow which is little more than MPLS-like label-based forwarding. One can think of the current OpenFlow as an unhappy medium between these two extremes: not general enough for the edge, and not simple enough for the core.

One might argue that OpenFlow’s lack of generality is appropriately tied to current hardware limitations, and that proposing a more general form of OpenFlow is doomed to fail. But at present much edge forwarding in datacenters is done in software by the host’s general-purpose CPU. Moreover, the vast majority of middleboxes are now implemented using general-purpose CPUs, so they too could implement this edge version of OpenFlow. More generally, any operating system supporting Open vSwitch or the equivalent could perform the necessary edge processing as dictated by the network controller. Thus, we believe that the edge version of OpenFlow should aggressively adopt the assumption that it will be processed in software, and be designed with that freedom in mind.

**Why is simplicity so important?** Even if one buys the arguments about the edge needing to become more flexible (to accommodate the generality needed in the host-network interface) and able to become more flexible (because of software forwarding), this doesn’t imply that it is important that the core become simpler. There are two goals to simplicity, reduced cost and vendor-neutrality, and the latter is probably more important than the former. Even if the additional complexity were not a great cost factor, if one is striving for vendor-neutrality then one needs an absolutely minimal set of features. Once one starts adding additional complexity, some vendors will adopt it (seeking a competitive advantage on functionality) and others won’t (seeking a competitive advantage on cost), thereby lessening the chance of true vendor-neutrality. We believe this is likely to happen with the emerging OpenFlow specifications, but would not apply to simple label-switching
What does this mean for networking more generally? Our design enables new functionality (whether it be forwarding models, middlebox processing, or new Internet architectures) to be introduced in software and then, if popular but inefficient, hardware support can be introduced to provide the necessary speed. This is how the virtualization of computation was introduced (i.e. the early software versions proved extremely useful but too slow for many uses, but then virtualization support was introduced into commodity CPUs to address this). This is in direct contrast with the canonical SDN approach, where new forwarding functionality must first be introduced into the OpenFlow specification and implemented in ASICs, and only then can it be made available to users.

If indeed we arrive at a point where the edge processing is done in software and the core in simple hardware, then the entire infrastructure becomes much more evolvable. Consider the change from IPv4 to IPv6; if all IP processing were done at the edge in software, then simple software updates to hosts (and to the relevant controllers) would be sufficient to change over to this new protocols. While we started focusing on infrastructure over architecture, this is one way in which an improved infrastructure would help deal with architectural issues.

---

7 Of course, vendors will always compete in terms of various quantitative measures (e.g., size of TCAM, amount of memory), but the basic interfaces should be vendor-neutral.
Chapter 4

Isolation of Consolidated Software Datapaths

Modern networks are replete with hardware middleboxes that perform a wide variety of functions, ranging from the generally applicable (e.g. firewalls, DPI) to the extremely specific (e.g. media gateways \([16, 26]\), broadband remote access servers). There is a growing discomfort with the cost of procuring and the difficulty of managing and deploying these specialized components. As a result, there is now an active movement towards Network Function Virtualization (NFV) which would replace special-purpose hardware devices with virtualized network functions (VNFs), typically running as a VM, that could run on racks of generic servers. The NFV approach offers many advantages, such as low cost, ease of deployment, and statistical multiplexing over a shared infrastructure (e.g., racks of servers).

However, there is one oft-overlooked disadvantage to this move to software. Because they had fixed functionality and dedicated single-purpose hardware, the physical incarnations of these functions had well-understood performance specifications, which allowed operators to offer SLAs to their customers. Once these functions move to software, providing such performance specifications is much harder. The difficulty in predicting comes...
largely from the fact that such VNFs would coexist on a single shared infrastructure; even if one understood how each VNF performed in isolation, one could not predict how they would perform when coexisting on the same machine.

One recent paper [23] provided great hope in this regard, by demonstrating a methodology that achieved high accuracy predictions for certain workloads and assumptions. This work recognized that the cache was the key shared resource for packet-processing functions, and that understanding how VNFs utilized the cache in isolation was enough to predict the performance of a set of VNFs sharing that cache. Unfortunately, as we will show, this performs significantly less well for newer hardware and workloads, and thus we must look elsewhere for a solution.

The conjecture driving this chapter is that a new hardware development, cache allocation technology (CAT), may provide exactly the tool we need to make better performance predictions for these virtualized network functions running on a shared infrastructure. CAT is nothing more than the ability to assign shares of the cache to cores. Such assignments can overlap (that is, two cores can partially share part of the cache), but cores can only use the portion of the cache assigned to them. We demonstrate that this is sufficient to provide much better predictability for these VNFs, and restores hope that carriers will be able to offer SLAs when providing tenants with virtualized network functions.

In the sections that follow, we first provide some basic background (Section 4.1) on the systems we consider. We then (Section 4.2), through extensive testing with a set of applications under a wide variety of scenarios, show that cache isolation is both necessary and sufficient for making performance predictions in this context. The necessary portion of this argument illustrates why the previous work [23] no longer applies, while the sufficient portion of our argument shows how CAT can enable reliable performance predictions regardless of the coexisting applications. However, this does not directly lead to efficient SLA enforcement, so in Section 4.3 we show how to use these performance predictions to enforce SLAs using close to the minimal number of machines. We discussion
related work in Section 4.4 and conclude in Section 4.5.

4.1 Background and Context

4.1.1 System Setup

<table>
<thead>
<tr>
<th>Processors</th>
<th>2 × Intel Xeon E5-2658 v3 @ 2.2GHz (24 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>L1/L2/L3: 32K/256K/30M</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4-2133 64GB/socket</td>
</tr>
<tr>
<td>NICs</td>
<td>3 × X520-Q1 (12 × 10Gbps ports)</td>
</tr>
</tbody>
</table>

Table 4.1: Test server specification.

We perform all packet processing on a state-of-the-art multi-core server. Figure 4.1 shows the high-level architecture of each server, and Table 4.1 summarizes its features. For our work, the most relevant features are the two Intel Xeon processors, each with 12 × 2.2GHz cores sharing a 30MB L3 cache and an integrated memory controller. Moreover, each server is equipped with 12 × 10Gbps interfaces, and there is sufficient bandwidth between the network interfaces and the processors to support all interfaces receiving and sending at full speed.

Our software stack consists mainly of Linux (Debian, kernel 3.18.3), the Intel data-plane development kit (DPDK, version 1.7) [38], and user-mode Click (release 2.0.1) [50]. DPDK is a set of libraries that enable data transfer between network interface cards (NIC) and applications with minimum kernel involvement (e.g. the data is directly copied from the NIC to the application’s user space). We chose the pair of DPDK and Click as a viable candidate for an NFV platform because it combines (DPDK’s) high performance with (Click’s) flexibility and ease of use. However, our results do not depend on any particular feature of DPDK or Click, and they should hold for alternative software stacks as well.

We run three kinds of packet-processing applications: (a) “Standalone applications” run as native Linux processes without DPDK support; (b) “DPDK applications” run as
Chapter 4. Isolation of Consolidated Software Datapaths

Figure 4.1: High-level architecture of the server with Intel Xeon E5-2658 v3 (Haswell series) processor. IIO and IMC stand for integrated I/O and memory controller. Each NUMA node has a single DIMM populated per channel for a total of 64GBs of memory per node. NUMA node 1 has no peripheral routed to other than the three 4Gbps NICs (over PCIe Gen3) that we use in our experiments. Node 1, which we solely use in our experiments, is isolated from the Linux scheduler. QPI is idle in all our experiments.

native Linux processes that benefit from DPDK; (c) “Click applications” run as Click processes with DPDK support. Most of our applications fall into the third category.

Each server runs multiple application instances. In our context, an “application instance” is a user-level process that is subject to a single SLA. For example, suppose a cloud tenant wants all the traffic entering its virtual network from the outside world to be processed by an Intrusion Detection System (IDS) with minimum throughput 10Gbps. In our context, this translates into a user-level process that receives all traffic entering the tenant’s virtual network from the outside world and performs the requested IDS; we refer to such a process as one “application instance.” Similarly, if the tenant specifies a pipeline containing a firewall followed by an IDS subject to a single SLA, then a single “application instance” containing both services is instantiated.

We configure our servers such that: (a) Each application instance is pinned to a single core; this setup has been repeatedly found to maximize performance by avoiding mandatory cache misses (that occur when one core passes a packet to another) [22, 72,
Chapter 4. Isolation of Consolidated Software Datapaths

23, 38]. (b) Each instance stores all its data in local memory (memory that is accessed through the local memory controller), in order to minimize memory access time. (c) Two instances never share the same core; core sharing forces instances to share L1 and L2 caches, significantly complicating SLA enforcement—an unnecessary complication under the realistic assumption that each application can saturate at least one core.

4.1.2 Cache Allocation Technology

A key technology for our work is Intel’s cache allocation technology (CAT), which organizes processing cores into “cache classes” and assigns some part of the last-level cache to each class. If we define one cache class per core, and we configure the cache partitions to be non-overlapping, CAT provides perfect last-level cache isolation across applications running on different cores. The support for cache monitoring and allocation is a newly introduced feature in a select number of Intel Xeon Haswell processor series.

Our current processors support only four cache classes (each with access to a subset of 20 cache ways). Given that each processor has 12 cores, when we enable CAT, we have to group three cores together into one cache class. This means that we cannot perfectly isolate our applications. Nevertheless, we still show significant benefits from using CAT. More importantly, the trend is towards rapidly increasing the number of cache classes, so this artificial sharing between cores within a cache class may no longer be necessary in the near future. Thus, we do not spend much time investigating the best strategies for sharing a cache class, and instead focus on how to utilize cache isolation.

4.1.3 Workloads

---

1The main motivation for the introduction of CAT was isolation across tenants in virtualized environments. In this context, it makes sense to have at least one cache class per tenant, and cloud providers already serve tens of tenants on each physical machine. Thus, it is a natural expectation that the number of classes will soon not be a future bottleneck for our purposes.
<table>
<thead>
<tr>
<th>Application</th>
<th>Platform/ I/O</th>
<th>Million packets per second</th>
<th>Cycles per instruction</th>
<th>L3 references per sec (millions)</th>
<th>L3 hit rate (percent)</th>
<th>Kilocycles per packet</th>
<th>L3 references per packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>Click/DPDK</td>
<td>4.43</td>
<td>1.79</td>
<td>24.66</td>
<td>99.93</td>
<td>0.50</td>
<td>5</td>
</tr>
<tr>
<td>STAT</td>
<td>Click/DPDK</td>
<td>2.88</td>
<td>1.47</td>
<td>37.16</td>
<td>99.90</td>
<td>0.76</td>
<td>12</td>
</tr>
<tr>
<td>NAT</td>
<td>Click/DPDK</td>
<td>2.07</td>
<td>1.48</td>
<td>36.40</td>
<td>99.99</td>
<td>1.06</td>
<td>17</td>
</tr>
<tr>
<td>RE</td>
<td>Click/DPDK</td>
<td>2.25</td>
<td>1.37</td>
<td>32.41</td>
<td>99.89</td>
<td>0.97</td>
<td>14</td>
</tr>
<tr>
<td>EndRE</td>
<td>Click/DPDK</td>
<td>1.82</td>
<td>1.47</td>
<td>25.47</td>
<td>99.85</td>
<td>1.21</td>
<td>13</td>
</tr>
<tr>
<td>FW</td>
<td>Click/DPDK</td>
<td>0.09</td>
<td>0.54</td>
<td>1.29</td>
<td>95.76</td>
<td>24.26</td>
<td>14</td>
</tr>
<tr>
<td>VPN</td>
<td>Click/DPDK</td>
<td>0.36</td>
<td>3.02</td>
<td>5.52</td>
<td>99.07</td>
<td>6.11</td>
<td>15</td>
</tr>
<tr>
<td>IDS</td>
<td>Native/Netmap</td>
<td>0.68</td>
<td>1.48</td>
<td>39.81</td>
<td>91.07</td>
<td>3.21</td>
<td>58</td>
</tr>
<tr>
<td>CLASS</td>
<td>Native/DPDK</td>
<td>0.78</td>
<td>1.38</td>
<td>0.96</td>
<td>99.76</td>
<td>2.82</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Workload characteristics
Each experiment typically involves two directly connected servers, one acting as an NFV platform and the other as a workload generator. The NFV platform runs multiple instances (typically one per core) of various realistic packet-processing applications, described below. The workload generator produces a traffic trace that consists of minimum-size packets from 100,000 different TCP or UDP flows. Packets are distributed across flows either uniformly or according to a Zipfian distribution with the distribution parameter 1.1. We experiment with minimum-size packets as this allows us to test at high packet rates [24].

One can categorize applications along two dimensions: the dominant mode (memory or CPU) and its cache behavior (sensitive and/or aggressive). We call an application “memory-intensive” or “CPU-intensive,” depending on whether it spends most of its cycles accessing the memory or performing other computation. We call an application “sensitive” and/or “aggressive” with respect to L3 cache sharing, depending on how such sharing may affect its own performance and the performance of other applications that it shares the cache with: when a sensitive application shares the L3 cache with an aggressive application, the latter causes the performance of the former to suffer, by frequently evicting its data. For example, a memory-intensive application that accesses non-cacheable objects is aggressive (because it brings data into the cache at a high rate), but not sensitive (because it does not suffer when it shares the cache with other aggressive applications).

We now summarize the functionality of our packet-processing applications, as well as their behavior with respect to potentially contended resources (the L3 cache and memory controller).

**IP forwarding (IP):** A Click application that implements standard IP forwarding. The forwarding table is a RadixTrie, populated with 128,000 entries, and we use the data structure and corresponding lookup algorithm that come with the Click distribution. IP is memory-intensive (it reads and updates the IP header of each incoming packet) and
benefits from caching (the forwarding table is cacheable). The forwarding table does not fit in the L2 cache, making IP both sensitive and aggressive with respect to L3 cache sharing.

Statistics collection (STAT): A Click application that implements IP plus per-flow statistics collection. In particular, it maintains a hash table that stores a packet counter and timestamp per active TCP and UDP flow, similar to Cisco’s NetFlow [17]. STAT is also memory-intensive, as well as sensitive and aggressive with respect to L3 cache sharing—more so than IP, because both the forwarding table and the hash table are cacheable but do not fit in the L2 cache.

Firewalling NAT gateway (NAT): A Click application with the MazuNAT configuration that comes with the Click distribution. It is a firewalling NAT gateway developed by Mazu Networks. This is a generalized version of network address translation (NAT), where any part of the IP- or transport-layer header may be rewritten (not just the IP source address and port number as in classic NAT), depending on the element’s configuration. It also supports additional features such as application-level packet rewriting (to support passive FTP) but those code paths are not exercised in our experiments. The data structure that maintains per-connection state is a sophisticated hash table with extensible per-key chains for handling collisions. NAT is also memory-intensive, as well as sensitive and aggressive with respect to L3 cache sharing.

Stateless firewall (FW): A Click application that implements STAT plus a stateless firewall that checks each incoming packet against an Access Control List (ACL) of 1000 rules. The ACL is a simple list that fits in the L2 cache. The ACL rules are chosen such that none of them match any incoming packet (hence, every packet ends up being compared against all the rules). FW is CPU-intensive; as a result, even though it accesses at least as many cacheable objects as IP and STAT per packet, it is significantly less sensitive or aggressive with respect to L3 cache sharing.

Virtual Private Network (VPN): A Click application that implements STAT
and then encrypts each incoming packet using the AES-128 algorithm. This is another example of a CPU-intensive application that is significantly less sensitive or aggressive with respect to L3 cache sharing than our first three types.

**Redundancy elimination (RE):** A Click application that implements STAT plus redundancy elimination as described in [77]. In particular, it maintains a “packet store” (a cache of recently observed content) and a “fingerprint table” (that maps content fingerprints to packet-store entries). For each incoming packet, it uses these two data structures to check whether the packet’s content overlaps significantly with the content of other, recently observed packets; if yes, instead of transmitting the packet as is, it transmits an encoded version that eliminates the overlapping content. RE is memory-intensive and may or may not benefit from caching, depending on the content redundancy of the particular workload. The packet store does not fit in the L2 cache, making RE aggressive and potentially (if there is content redundancy) sensitive with respect to L3 cache sharing.

**Improved redundancy elimination (EndRE):** A Click application that has the same goal as RE, but uses an alternative algorithm for redundancy elimination, contributed by [2]. The key difference is that we have a separate packet store per connection—the rationale being that packets from the same connection are more likely to have overlapping content.

**Traffic classification (CLASS):** A DPDK application that implements the Effi-Cuts [90] traffic-classification algorithm. We configure it with 32 000 classification rules generated with ClassBench [83]. All the rules fit in the L2 cache, making CLASS CPU-intensive and neither sensitive nor aggressive with respect to L3 cache sharing.

**Intrusion Detection (IDS):** The Snort [69] intrusion-detection system, running as a standalone application, configured with the “Emerging Threats” ruleset [25] and the “VRT” ruleset [84] for the overall 20872 detection rules. With these active rules and our sample input traffic used throughout the experiments, Snort’s working memory set
is larger than the whole last-level cache. Our Snort setup uses netmap [68], a framework for high-speed packet I/O, to avoid the overhead of native drivers.

We also use two synthetic applications for profiling and to exercise specific server bottlenecks:

**SYN**: A Click application that performs a configurable number of CPU operations and read a configurable number of random memory locations from a data structure. We use this application to evaluate the accuracy of performance degradation prediction algorithm in [23].

**STREAM**: We use Intel’s Memory Latency Checker (MLC) [39] to induce contention on the memory controller. The number of idle CPU cycles between successive memory operations could be configured to generate different load levels. We use MLC to evaluate the memory controller sensitivity of our networking pipelines.

### 4.2 Cache Isolation

In this section we address three questions: (a) is cache isolation necessary?, (b) is cache isolation sufficient, and (c) how much does cache isolation cost (in terms of performance)?

#### 4.2.1 Cache Isolation is Necessary

In the context of a multi-core server running multiple application instances, contention for the last-level cache can severely degrade application performance [40, 18], and this holds especially for packet-processing applications [23]. A packet-processing application typically reads each incoming packet from memory, performs (often simple) computations like checksum update or TTL decrement, looks up content in (often cacheable) data structures like forwarding or filtering tables, and writes the packet back to memory. When two such application instances $A$ and $B$ share the last-level cache, they may affect each other’s performance, depending on the relative frequency with which they (a) access
cacheable objects and (b) bring new content into the cache. For example, if $A$ and $B$
both perform our IP application, they access cacheable objects (the forwarding table) at
the same rate, and they bring new content into the cache (a new packet or previously
evicted forwarding-table entry) also at the same rate. Hence, each of $A$ and $B$ is both
sensitive and aggressive with respect to cache sharing, and they suffer equal performance
degradation.

Predicting the performance of a set of applications sharing the cache requires predict-
ing the effects of cache sharing. In the past, researchers tried to construct mathematical
models that would enable such prediction [1, 79, 86], but we are not aware of any such
model that was robust enough to application idiosyncrasies and hardware changes to find
its way into practice.

Figure 4.2: Absolute and percentage error of the degradation prediction method proposed
in [23] under various settings.
Recently, Dobrescu et al. proposed a simple prediction technique that does not involve any mathematical modeling [23]: Consider an application instance $T$ that achieves a given performance when running on a single core alone (without any other instances running).\footnote{In our experiments, we affinitized all threads of an application to a single core, but for multi-threaded applications, we could extend our profiling to use multiple cores with a fixed thread-to-core mapping (that the SLA enforcer needs to use).} Suppose we want to predict $T$’s performance drop if it shares the cache with $n$ competing application instances $C_1 \ldots C_n$. According to the proposed technique, we can do this as follows: (a) First, we estimate the aggregate number of cache references per second (say $x$) that the competing instances $C_i$ will perform when sharing the cache with $T$. (b) Second, we measure $T$’s performance drop (say $d$) when it shares the cache with $n$ SYN application instances that perform an aggregate number of $x$ cache references per second. (c) We predict that $T$’s performance drop will be equal to $d$—the assumption being that the $C_i$ instances and SYN instances will cause similar damage to $T$.

The intuition behind this approach is that the dominant factor that determines an instance’s performance drop is the aggregate rate of cache references performed by the competing instances; the particular access pattern (which is what the earlier mathematical models tried to capture) is relatively insignificant.

We experimented with this technique, and Figure 4.2 shows its error in predicting performance drop in various scenarios. Each bar shows the average difference between predicted and observed performance drop suffered by a target application instance $T$ when sharing a processor with 5 identical competing application instances (for an overall of 9 datapoints per target application) similarly to the choice of competitors in [23]. For example, the tallest bar in Figure 4.2a shows the average of the technique’s absolute error\footnote{The absolute error is in percentage points because performance drop itself is measured in percentage points. For example, if the technique predicted performance drop 10%, but the actual performance drop was 5%, then the absolute prediction error is 5%.} in predicting the performance drop of a STAT instance when sharing the L3 cache with 5 identical competing instances, under Zipfian traffic and disabled DDIO; the tallest bar in Figure 4.2b shows the technique’s relative error in predicting the performance drop...
of a NAT instance when sharing the L3 cache with 5 identical competing instances, under Zipfian traffic and enabled DDIO. Even though our servers have 12 cores per processor and 30MB of last-level cache, we ran these experiments with 6 competing instances and 19.5MB of last-level cache, in order to stay as close as possible to the experimental setup used in [23].

We found that, in our setup, this technique is much less accurate than in the setup used in [23]: we measured absolute error up to 13.2% and relative error up to 78.7%. Our intuition on these results is the following: Prediction would be perfect, if the actual competing instances behaved exactly like SYN instances. However, a realistic application typically exhibits more temporal locality in memory accesses than SYN (which performs random accesses over a data structure that is larger than the L3 cache). For example, consider an IP application instance receiving uniform traffic; even though the IP instance accesses the forwarding-table entries with a random pattern (due to traffic uniformity), it still accesses the root node of the RadixTrie for every received packet. As a result, a realistic application instance typically causes less damage than a SYN instance that performs the same cache references per second, because more of its references are to objects that are already cached (hence do not cause any evictions).

Our position is that we should not try to predict the effects of cache sharing, for two reasons:

First, it is hard to design a robust prediction algorithm, not only because of the complexity of the applications themselves, but because application behavior depends on constantly evolving hardware optimizations and software stacks. For example, DDIO is a hardware optimization that brings incoming packets directly into the last-level cache, before an application instance reads them for the first time. Figure 4.2 shows that the same prediction technique yields significantly different results with and without DDIO (enabling DDIO decreases prediction error for most applications, but it increases prediction error for NAT). Moreover, the results in Figure 4.2 are significantly worse than
those presented in [23], even though the experimental setup is similar; this could be either because of subtle differences in subsequent generations of Intel Xeon processors, or because of differences in the software stacks (they used kernel-mode Click, whereas we use the user-mode Click that we patched to perform I/O using Intel DPDK).

Second, even a robust prediction algorithm for a given scenario does not enable accurate prediction in operational settings: To predict the effects of cache sharing on a set of application instances $T_i$, any prediction algorithm requires as input (a) the configuration of each $T_i$, e.g. the size of its data structures and (b) information about the traffic that each $T_i$ will process, e.g. packet size and flow distributions. Either of these can have a dramatic impact on $T_i$’s sensitivity and aggressiveness. For example, an IP application instance can go from zero to very high sensitivity, if its flow-size distribution changes from very skewed to uniform. Or, a FW application instance can go from very high to zero aggressiveness, if its ACL is initially empty (in which case FW behaves like the memory-intensive STAT) and then populated with a few thousand rules (in which case FW becomes CPU-intensive). It is hard to imagine a multi-tenant environment that keeps track of configuration and incoming-traffic changes of all application instances in real time.

The bottom line is that if we cannot predict the effects of cache sharing, we need to eliminate cache sharing in order to enforce SLAs. Furthermore, in Section 4.3, we show that cache sharing indeed results leads to SLA violations. Thus, we need cache isolation across application instances.

### 4.2.2 Cache Isolation Is Sufficient

Since the last-level cache is not the only shared resource, the next question is whether cache isolation *alone* makes application performance predictable, or whether we also need to consider the memory controller—the other resource shared among cores within the same NUMA domain.
To answer this question, we first ran each application in various scenarios and measured the percentage difference between the minimum and maximum throughput observed. Figure 4.3 shows the results; for the moment, we will concentrate on the “Isolated-Realistic” and “Shared-Realistic” labels, and ignore “Isolated-STREAM.” Each bar shows the throughput variation of a target application instance when sharing an L3 cache with 11 competing instances, with cache isolation (“Isolated-Realistic”), and without cache isolation (“Shared-Realistic”). In all experiments, both the target instance and the competing instances run realistic packet-processing applications. In all experiments with cache isolation, the target application instance is exclusively assigned 10% of the L3 cache (the smallest size CAT allows us to assign), while the remaining 90% is shared by the competitors. We chose this particular cache partitioning in order to maximize the sensitivity of applications to the memory controller; this is because with a smaller cache, applications incur higher L3 cache misses and trips to memory. For each target application instance, we ran 9 experiments where all competing application instances were of the same type. This choice of competitors ensures that the least and most aggressive competitors are included in our evaluation.

We found that cache isolation eliminates most of the performance variation due to processor sharing: we measured maximum variation in performance up to 2.9% with cache isolation and up to 42.9% without cache isolation. For example, the right-most
Figure 4.4: Degradation of various packet processing pipelines and memory controller latency as a function of load on the memory controller. They track each other very closely.

The bar in the figure shows the difference between the best and worst throughput observed by a NAT application instance when competing for resources due to processor sharing is 2.9% with cache isolation, and 40.3% without cache isolation.

These results are surprising, because some of our realistic packet-processing appli-
cations are memory-intensive, we expected their performance to be quite sensitive to contention for the (shared) memory controller.

To understand this insensitivity of our applications to competition for the memory controller, we measured application performance drop as a function of memory-controller load. Figure 4.4a shows the results. Each curve shows the performance drop suffered by a target packet-processing application instance, when sharing a processor with 11 STREAM instances. In all experiments, the target instance is exclusively assigned 10% of the L3 cache, while the remaining 90% is shared by the competing STREAM instances. To complement this data, Figure 4.4b shows memory-access latency as a function of memory-controller load.

We found that memory-controller load affects application performance only when it reaches 20%, and even then it causes performance drop up to 5%; this behavior changes when memory-controller load reaches 65%, at which point we observe the start of congestion collapse (Figure 4.4a). Memory-access latency follows exactly the same pattern as application performance drop (Figure 4.4b). These results indicate that, at least in a state-of-the-art Xeon processor, as long as we keep memory-controller load below the point where congestion collapse begins (which is anyway a reasonable practice), sharing the memory controller does not introduce significant performance drop. This condition was true in all experiments that we ran with our realistic packet-processing applications, which explains why processor sharing with cache isolation caused relatively small performance drop (Figure 4.3, “Isolated-Realistic” data): Across all experiments we run with 12 realistic applications, the memory controller utilization remained below 17%.

In a server running packet-processing applications, overloading the memory controller is highly unlikely. Even the trip to memory to read and write packets for I/O to the NIC is no longer needed thanks to direct I/O to the last-level cache by DDIO.

We should note that this observation does not hold for a server running general (non-packet-processing) applications. To illustrate, we add to Figure 4.3 a third set of data
Figure 4.5: Cache sensitivity of various pipelines under uniform traffic. The data points below 3MB are approximated.

points, labeled “Isolation-STREAM.” Each data point is the performance drop of a target packet-processing application when it shares the memory controller with 11 STREAM instances accessing the memory at the highest possible rate. In all experiments, the target packet-processing instance is exclusively assigned 10% of the L3 cache, while the remaining 90% is shared by the competing STREAM instances. We see that performance drop is as high as 35.3%, even though we are using cache isolation. This is because the competing STREAM instances are loading the memory controller beyond 65%, which causes a significant increase in memory-access latency, hence a significant drop in the performance of any memory-intensive application instance.

Thus, we find that for packet processing applications sharing a cache, CAT is sufficient to provide predictable performance.
4.2.3 The Cost of Cache Isolation

The remaining question is how much performance cost does CAT incur. It makes sense that isolation leads to predictable performance (though this depended on the cache being the main bottleneck, which is true for packet-processing applications but not for all applications). But does this predictability come with a performance penalty compared to running without CAT? This is what we address in this section.

First, we measured application sensitivity to cache size. Figure 4.5 shows the results. Each curve shows how the performance of a target application instance drops as we decrease the size of the L3 cache partition that the instance can access. To generate each curve, we ran multiple experiments where the target instance runs alone, with cache isolation, and we varied the size of the accessible L3 cache partition across experiments. Not surprisingly, reducing the accessible cache size can have a dramatic effect. As a side-note, the figure indicates the working-set sizes of our applications, e.g. IP performance drops only when the accessible cache size decreases below 10MB; IDS performance starts dropping immediately, indicating a working set that is larger than the total L3 cache.

Intuitively, the cost of cache isolation depends on how we partition the cache. For example, suppose we have a STAT instance and three FW instances sharing the same processor. Recall that STAT is both sensitive and aggressive with respect to cache sharing, whereas FW is neither. Hence, if we do not use cache isolation, and the four instances share the L3 cache, neither will suffer any significant performance drop. If we do use cache isolation, and we split the cache evenly between the four instances, then the performance of the STAT instance will drop by 32% (Figure 4.5, cache size 7.5MB). However, if we allocate 18MB to the STAT instance and 3MB to each of the two FW instances, then none will suffer any significant performance drop—making the cost of cache isolation zero.

We considered two ways of partitioning the cache: (a) so as to minimize the maximum performance drop of any single instance (“minmax” objective), and (b) so as to minimize
Figure 4.6: Overhead of full last-level processor cache isolation.
the aggregate performance drop of all the competing instances ("minsum" objective).

Next, we assessed the minimum cost of cache isolation as follows: We considered all possible combinations of 4 application instances, where each instance can run any of the 6 applications featured in Figure 4.5 (a total of 1296 combinations). For each combination, we considered the scenario where the 4 application instances share a processor, with and without cache isolation; in case of cache isolation, we computed the optimal partitioning according to the minmax and minsum objectives. Figure 4.6 shows the minimum "isolation penalty," i.e. the gap between application performance under cache sharing on the one hand, and application performance under cache isolation with optimal partitioning on the other. Figure 4.6a plots average isolation penalty per application (each boxplot summarizes results across all experiments), while Figures 4.6b and 4.6 plot isolation penalty per application instance (each boxplot summarizes results across one application).

We found that a few instances suffer an isolation penalty up to around 30%, but most instances enjoy a negative isolation penalty, i.e. their performance is better with cache isolation than cache sharing; this is true for both partitioning objectives. The intuition is the following: Most of our applications are both sensitive and aggressive with respect to cache sharing, but to different degrees. Suppose four equally sensitive instances share a processor, and one of them is more aggressive than the rest. Cache sharing allows this extra aggressive instance to consume more of the cache at the expense of the other three. In contrast, cache isolation splits the cache evenly across the four instances, thereby improving the performance of the three instances at the (modest) expense of the extra aggressive instance.

To summarize, the question of the performance cost of CQoS is subtle, because it depends on how you allocate the cache. There are certainly bad ways to allocate the cache that seriously degrades performance, but the question is whether there are good ways to allocate the cache so that the performance penalty is small (or even negative).
We found that in most cases we could allocate the cache to minimize the penalty.

4.3 Meeting SLAs

Our results so far indicate that cache isolation can help predict performance, but shed little light on how this ability can be used in practice. In this section, we describe our goal (§4.3.1), specify it formally (§4.3.2), and evaluate how well we can achieve it (§4.3.3).

4.3.1 Goal and Assumptions

Our over-riding goal is to allow network operators to reliably and efficiently provide tenants with SLAs, without having to measure how different applications interact when sharing the same physical server. We envision a scenario where each SLA has the form of a \{packet-processing application, rate\} tuple. For example, a tenant may require VPN encryption/decryption at a minimum rate of 1Gbps, or IDS processing at a minimum rate of 10Gbps. If a tenant requires a combination of applications at some target rate, e.g. IP forwarding plus IDS processing at rate 10Gbps, then the SLA tuple is \{IP+IDS, 10Gbps\}, i.e. IP+IDS becomes a new application type. We want to enable an operator to meet as many such SLAs as possible while using the minimum number of physical servers.

In meeting this goal, we assume that all the traffic of a given tenant that needs to be subjected to a given application can be broken into multiple streams (e.g. by a load-balancer) and sent to separate instances of that application. The aggregate rate of these streams must be greater or equal to the rate specified in the corresponding SLA, but the rate of each individual stream can be significantly less. We think that this is a reasonable assumption, given that load-balancing is already widely used in multi-tenant environments. For certain applications (e.g. IDS), we would need to ensure that any separation into streams is subject to certain constraints, e.g. all traffic to the same
destination address is assigned to the same stream (modern load-balancers can typically satisfy such constraints). In the results we present below, we do not consider such constraints.

Moreover, we make the following assumptions about the shape of application throughput as a function of (last-level) cache size: (a) There exists a “lower knee-point,” i.e. a minimum cache size below which application throughput drops exponentially. In our servers, this lower knee-point occurs at 10% of the L3 cache or less for all applications. CAT does not allow us to allocate less than 10% to any cache class.\(^4\) (b) For each application, there exists an “upper knee-point,” i.e. a maximum cache size beyond which application throughput improves insignificantly. Hence, we do not allocate more than this maximum cache size to any instance of this application. (c) Between the two knee-points, we can approximate application throughput with a linear function. These assumptions certainly hold for all the realistic packet-processing applications that we considered, and they allow us to model our problem as an integer program with linear and quadratic constraints, as we describe below.

### 4.3.2 Problem Statement

In our formulation, an SLA is for a single application type, and it defines a target rate promised to the tenant to whom the SLA belongs. To satisfy each SLA, we need to run one or more instances of the corresponding application, with each instance running on a single core alone, i.e. without sharing that core with any other instance (§4.1.1). For each SLA, we provide the solver with the following constants: (a) \(T_i\) is the target rate, (b) \(M_i\) is the maximum cache size to allocate to any one instance, and (c) \(\text{Coeff}_{i0}, \text{Coeff}_{i1}\) are the coefficients for the linear approximation of throughput as a function of cache size (§4.3.1). The maximum cache size and the coefficients are, of course, the same for all applications, the knee is not visible in Figure 4.5, but since the throughput drop at origin is 100% for all applications it follows that there is a sharp increase in drop as we approach the origin for all of them.

\(^4\)For most applications, the knee is not visible in Figure 4.5, but since the throughput drop at origin is 100% for all applications it follows that there is a sharp increase in drop as we approach the origin for all of them.
SLAs that refer to the same application.

Listing 4.1 shows our integer program, whose objective is to minimize the number of machines needed to guarantee all SLAs. We define the following variables:

- Integer variable $C_{ij}$ indicates the amount of cache allocated to instance $j$ of SLA $i$. Each such variable can take a value between 2 and 20 cache ways.

- Binary variable $N_k$ indicates whether machine $k$ is active or not.

- Binary variable $I_{ijk}$ indicates whether the instance $j$ of SLA $i$ is active on machine $k$.

Our constraints ensure that: (a) an instance runs on at most one machine; (b) there are at most NodeCores instances running on each machine with NodeCores cores; (c) the total amount of cache used by instances running on one machine is less than or equal to the machine’s total cache size NodeCache; (d) the aggregate throughput of an SLA’s instances is greater or equal to the SLA’s target rate; (e) a machine is active when there is at least one instance running on that machine (this constraint is formulated using two inequality constraints). Listing 4.1 formalizes these constraints in the same order as we described them above.

In our implementation, we also added constraints to speed up the solver, e.g. we add an upper bound on the number of instances per SLA (computed assuming each instance gets the minimum cache allocation), and a lower bound on the number of machines (computed ignoring the per-machine cache limitation).

Even though our formulation assumes identical machines, it is straightforward to extend it to a heterogeneous set. For that, we would need to provide the solver with a per-machine throughput function for each unique application type, along with the L3 cache size and number of cores per machine.

As a side-note, it is also possible to extend our formulation to support SLAs with latency constraints. For that, we would need to provide the solver not only with through-
Variables: $C_{ij} \in \{2, \ldots, M_i\}$
$N_k \in \{0, 1\}$
$I_{ijk} \in \{0, 1\}$

Minimize: $\sum_{k \in \text{Nodes}} N_k$

Subject to: $\forall i, j : \sum_k I_{ijk} \leq 1$
$\forall k : \sum_{i,j} I_{ijk} \leq \text{NodeCores}$
$\forall k : \sum_{i,j} C_{ij}.I_{ijk} \leq \text{NodeCache}$
$\forall i : \sum_{j,k} I_{ijk}.[\text{Coeff}_{i0}.C_{ij} + \text{Coeff}_{i1}] \geq T_i$
$\forall k : \sum_{i,j} I_{ijk} - N_k.(\text{NodeCores}) \leq 0$
$\forall k : \sum_{i,j} I_{ijk} - N_k \geq 0$

Listing 4.1: An integer program the solution to which is an optimal placement and cache allocation with minimum number of machines that satisfies the requirements of all the given SLAs.
put functions, but also with information about latency as a function of cache size and incoming traffic rate.

4.3.3 Experimental Evaluation

With the above program at hand to compute near-optimal instance placements and cache allocations, we sought to quantify the benefit of cache isolation as a mechanism to guarantee SLAs. A key question is to what we should compare our program’s output, and our first thought was to compare it to an optimal instance placement that guarantees the same SLAs but without cache isolation. However, even though such an optimal placement definitely exists, there is no evidence that a practical program can find it (§4.2.1): First, there exists no algorithm that accurately predicts the performance drop suffered by application instances when sharing the same processor. Second, even if such an algorithm did exist, it is unclear how we would deal with changes in the configuration or incoming traffic of application instances—whereas, with cache isolation, a badly behaving application instance (e.g. with transiently abnormal traffic) can only violate its own SLA.

Instead, we quantified the benefit of cache isolation indirectly, in the following way: We used our program to compute a near-optimal instance placement in various scenarios, then counted the number of SLA violations that we would see with the same instance placement but without cache isolation. For example, suppose our program computes an instance placement that guarantees 100 SLAs using 10 machines; we count the number of SLA violations that we would see with the same instance placement if the 10 machines did not implement cache isolation. If we count no SLA violations, this means that cache sharing is equally or perhaps more efficient than cache isolation (it can guarantee the same SLAs with as many or perhaps fewer machines). Conversely, the more SLA violations we count, the bigger the benefit of cache isolation.

As another metric of the efficiency of our approach, we compared the number of
machines that we need in order to meet a set of SLAs with cache isolation (according to our program) on the one hand, to the number of machines that we would need if our servers had unlimited L3 cache on the other. We compute the latter number as $\left\lceil \frac{\sum X_i(M_i)}{\text{NodeCores}} \right\rceil$, where $X_i(M_i)$ is the throughput of instance $i$ given its maximum cache allocation $M_i$. This simply says that, assuming unlimited L3 caches, every instance could run on with its maximum cache allocation.

To understand which applications benefit the most from cache isolation, we generated three sets of SLAs, each using a different set of applications: (a) cache-sensitive applications (STAT, IP, NAT, RE), (b) cache-insensitive applications (CLASS, FW, VPN), and (c) all the applications we have. We generated between 70 and 80 SLAs and set the target rate for each SLA to the highest rate that a single instance of the corresponding application can sustain when it has the entire L3 cache to itself. We set the number of cores per processor to four, because our current hardware supports only four cache classes, and our approach is to exclusively assign a cache class to each core; this limitation will be removed in upcoming hardware (§4.1.2). To solve our integer program, we used Gurobi [33] running on eight 2.6GHz cores; it found solutions with 20% optimality gap for the cache-sensitive and cache-insensitive application sets, and with 28% optimality gap for the full application set, all within 3 minutes. When we say that an SLA is violated by $X\%$, we mean that the aggregate throughput of the SLA’s instances was $X\%$ less than the SLA’s target rate.

Table 4.3 shows our results. In all cases, cache isolation guarantees all SLAs, while cache sharing leads to significant SLA violations. Given a cache-sensitive application set (first row), to meet 80 SLAs with cache isolation, we need 115 cores in a total of 29 machines; when we use the same instance placement without cache isolation, 46.25% of the SLAs are violated by more than 5%, and 6.25% of the SLAs are violated by more than 20%. Given the full application set (third row), to meet 72 SLAs, we need 100 cores in a total of 25 machines; when we use the same instance placement without cache
isolation, 19.44% of the SLAs are violated by more than 5% and 1.38% of the SLAs are violated by more than 20%. Finally, on average, our approach needs about 30% more machines than what it would need if our servers had unlimited L3 caches (“Machines lower-bound” column)—an indication of the cost we pay due to the cache sensitivity of our applications.

To conclude, we find that cache isolation can not only guarantee SLAs for realistic packet-processing applications, but—surprisingly—it can do so significantly more efficiently than cache sharing.
<table>
<thead>
<tr>
<th>Application types</th>
<th>SLAs required</th>
<th>Cores required</th>
<th>Machines required</th>
<th>Machines lower-bound</th>
<th>Violations (percent)</th>
<th>&gt; 2%</th>
<th>&gt; 5%</th>
<th>&gt; 10%</th>
<th>&gt; 15%</th>
<th>&gt; 20%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitive</td>
<td>80</td>
<td>115</td>
<td>29</td>
<td>20</td>
<td>52.50</td>
<td>46.25</td>
<td>25</td>
<td>11.25</td>
<td>6.25</td>
<td></td>
</tr>
<tr>
<td>Insensitive</td>
<td>78</td>
<td>94</td>
<td>24</td>
<td>20</td>
<td>15.38</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>72</td>
<td>100</td>
<td>25</td>
<td>18</td>
<td>34.72</td>
<td>19.44</td>
<td>13.89</td>
<td>8.33</td>
<td>1.39</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Resources needed by our approach to meet our SLAs with cache isolation (columns 2 and 3. Resources needed by our approach assuming unlimited L3 caches (column 4). SLA violations resulting from the same instance placement as produced by our approach but without cache isolation (columns 5–9).
4.4 Related Work

The most closely related work (aside from [23] which was previously discussed) is DRFQ [29], which models a packet-processing platform as a pipeline of resources, where each packet is sequentially processed by each resource. DRFQ’s primary goal is to provide per-flow fairness while we seek to guarantee SLAs. Ignoring that memory subsystem and CPU are not independent resources, one may be able to extend DRFQ to satisfy SLAs: Given our total resources and each user’s dominant resource, we can determine what SLA we can promise to each user, translate each SLA into a given fraction of a dominant resource, and use a DRF scheduling algorithm (with weights) to enforce each SLA. However, the moment we consider the cache, the relationship between SLAs and fairness stops being straightforward: (a) Without cache partitioning, we cannot assign a fraction of the cache to a given user. (b) Even with cache partitioning, we do not know how to translate each SLA into a given fraction of the cache. (c) Because the cache is so tightly coupled with other resources, we cannot use a DRF scheduling algorithm to enforce our SLAs. Our work addresses (a), (b), and (c), i.e. shows how to translate SLAs into resource allocations when one of the resources is the cache.

Even though CAT just recently found its way to production commodity processors, the mechanisms and use cases of cache partitioning have been studied in the past decade [41, 18, 40]. We showed that the primary contended resource for competing packet-processing pipelines is the last-level cache and took first steps toward quantifying possible benefits of cache partitioning in the software forwarding domain.

Contention on the shared last-level cache could be controlled without hardware support (e.g., page coloring [81, 96]). With hardware support, we have an accurate knob to control cache occupancy without any overhead for cache reallocation but at a coarse granularity. Future work could use a hybrid approach to provide further isolation at a finer granularity.

A closely related line of research is contention-aware scheduling. Some work [8] pro-
poses to locally schedule threads so as to minimize resource contention, while other work [21, 54] characterizes datacenter and cloud workloads offline or in real time in order to enable cloud-scale contention-aware scheduling. We assume that packet-processing applications are available prior to instantiation and can be profiled with representative configuration and traffic. If either assumption does not hold, then we need to do dynamic profiling [21, 18] and adjust our cache allocations in real time [18].

4.5 Concluding Remarks

This chapter started with the observation that the move to software packet processing on shared infrastructures, as in NFV, has wide support, but there has been little work in assuring that such environments can support SLAs. We first addressed previous work which had made an extremely promising start on this problem [23]. While the initial results from their approach were encouraging, when we revisited their technique with more modern hardware and more strenuous use cases, we found it impossible to predict the performance of a packet processing application when it is running on the same machine as unknown other packet processing applications. Such performance prediction is necessary for providing SLAs, so we turned to CAT as a possible remedy for these problems. Our subsequent work revolved around four questions:

Is CAT necessary to make accurate performance predictions? Here, our answer was affirmative, based on extensive evaluation of the previous approach to performance prediction.

Is CAT sufficient to make accurate performance predictions? We found that the answer was yes, but this is because the memory controller is not a bottleneck for packet-processing applications (which is not true for all applications).

Does CAT impose performance penalties? This is not a simple question, but we found that with intelligent cache allocation schemes the performance cost of CAT was mostly
minor (with some rare exceptions).

*Can one use this to build a system that enforces SLAs?* We found that we were able to provide SLA enforcement with these techniques.

Our choice of execution model (borrowed from a line of recent research on software packet processing [24, 22, 23]) sidesteps many sources of unnecessary contention. It simplifies the performance isolation problem enough that control over last-level processor cache becomes the only knob necessary to control interference. We acknowledge that the primary enabler of our simple solution is the advances in Intel processors over the past few generations that target network-heavy workloads.

While this work does provide some hope for providing SLAs, there is still much to be done. For instance, we did not consider the impact on packet processing latency. Conceptually, we could extend our techniques to measure latency as a function of utilization for different cache allocations, and take that into account when computing our SLA-conforming placements and cache allocations. However, we have not yet measured how this would work in practice.

In addition, we have not run this in an operational setting where traffic and policies are changing over time. This would be the next step in turning our static predictions into a more robust system. The key point, though, is that our work does show that CAT protects applications from variations in other application traffic and configuration. The open question is how to predict an application’s cache requirements when its own traffic and/or configuration are changing.
Chapter 5

Conclusion

This thesis offers a way for SDN networks to offer performance guarantees to their tenants. At a high-level, we achieved this goal in three steps. First, we showed, by example, that logically central control platforms could be built that scale and achieve the throughput and latency requirements of large networks (Chapter 2). Second, we presented an alternative SDN design that embraces the notions of fabric and software edge: fabric decouples packet delivery (implemented in the network core) from other functions (implemented at the network edge) and software edge (with the possibility of hardware assist) provides the much needed flexibility in data plane (Chapter 3). This design decouples lets individual functions be provisioned independently. Third, we study the performance degradation of a wide range of software packet processing pipelines under resource contention, and show that isolating last-level processor cache (among other shared resources) is sufficient to isolate the performance of consolidated pipelines (Chapter 4).

NOX-MT [88] and HyperFlow [87] were among the first works that looked at logically central control platform performance and scalability (Chapter 2). We demonstrate that NOX’s (the original SDN controller) poor performance was an implementation artifact and not an inherent architectural deficiency of SDN. A single NOX-MT instance can handle more than 400k requests per second (compared to 30k in NOX) with sub-2 mil-
lisecond response time (compared to 10ms in NOX) on a single core. We developed NOX verity based on NOX-MT. With HyperFlow we presented a distributed control platform that taps into the event-driven nature of NOX to transparently replicate the control state across multiple controllers. It does so by capturing events locally and replaying them on other controllers. Our implementation can replicate 1000 (state-altering) events per second across all controllers with a latency of one round trip time. Therefore this design scales well for networks and control planes in which a large fraction of network events do not alter the network-wide view. We have left characterizing the relationship between the inconsistency window size and the rate of network changes to future work.

We argued that SDN needs to change and learn from the lessons of MPLS and NFV (Chapter 3). We present an alternative design that embraces edge/core distinction and software edge. The core of the network (fabric) implements packet delivery in hardware but the rest of network functions are implemented at the edge in software. The fabric implements a simple interface to provide QoS-enabled packet delivery while completely hiding its implementation internals from the edge. The software edge provides the much needed flexibility in data plane, as well as an evolutionary path to hardware (e.g., NICs, coprocessors, extension to processor instruction set) for novel functions. The modularity that this design introduces in data plane enables us to provision resources for different network functions independently.

Lastly, we proposed a method to isolate the performance of software packet processing pipelines each running on a single processing core on the same server. We studied a wide-range of software packet processing pipelines, and showed that to guarantee performance isolation among colocated pipelines it is sufficient to isolate accesses to the last-level processor cache (among other shared resources). This is a promising result since support for last-level processor cache allocation is emerging in the commodity processors (first introduces in some Intel Haswell Xeon processors). We formulated the network SLA enforcement problem as an integer program the solution to which determines the instance
count for each pipeline, as well as placement and cache allocation for each instance.

5.1 Future Work

The lack of flexibility in the canonical SDN’s data plane (i.e., fixed simple datapath) was a pragmatic comprise. In this thesis we embraced software packet processing (with hardware assist) to offer full flexibility and edge/core distinction to offer functional modularity in datapath (Chapter 3. There are, however, intermediate design points that we did not consider like networks built with more flexible hardware switches like reconfigurable match table (RMT) switch [9] and Intel’s FlexPipe [63]. Offering guaranteed performance in other settings remains an open question.

Our software packet processing performance isolation work, presented Chapter 4, has a very rigid set of assumptions. We plan to relax those assumptions and generalize our work in the future. Below we summarize potential directions for future research.

Flexible pipeline placement. Due to operational considerations or simply to achieve better performance, elements of a packet processing pipeline may be placed on the same core, different cores on the same processor, different processors on the same machine, or even different machines altogether. Similarly, elements of multiple pipelines may be consolidated on a single processing core when demand is low. In Chapter 4, we assumed that all constituents of a pipeline are placed on a single core. When elements are placed on separate machines or processors the only thing that changes is that the network or processor interconnect are utilized. Given there is enough bandwidth, isolating last-level processor cache is sufficient to isolate elements from one another. Coplacement on different cores on the same processor, however, may introduce interference because packets are directly fed from a core to another. Coplacement of multiple pipelines on the same core would results in contention for the core resources (e.g., cycles, L1 and L2 caches). We leave characterizing and controlling degradation in these new cases to future
Isolation and performance trade-off. In Chapter 4, we only considered full cache isolation: no two pipelines have access to the same region of the last-level cache. However, there may be a performance advantage to sharing cache for some pipelines; for instance, two cache insensitive pipelines could be coplaced in the same class (with minimal cache allocation) freeing up cache space for other pipelines. Moreover, there is no other choice but to share cache among some pipelines where the number of supported non-overlapping cache classes is smaller than the number of cores. There are many ways to share cache among pipelines each with different level of performance and degree of isolation. We leave studying the trade-off and the choice of optimal overlapping allocation to future work.

Performance profiles. A key input to the model to compute SLA-satisfying placements and cache allocations (Section 4.3) is the performance profiles of pipelines as a function last-level cache size. In our work, we only looked at throughput as a performance metric. In future we plan to extend our work to look at other performance metrics (most importantly latency). Moreover, performance profiles are sensitive to input traffic and configuration for each pipeline. We leave devising a meta-profile for each pipeline that is independent of the input traffic and/or configuration to future work.

Realtime placement and resource allocation. We formulated the SLA problem as an integer program and used a generic solver to get a near-optimal solution. For very large problem sizes it may take significantly longer to get an acceptable solution. Even for modest problem sizes it takes the solver in the order of minutes to reach a solution. More importantly the time it takes to find a placement and cache allocation should be significantly less than the period in which configurations and traffic conditions and are stable. Therefore it is crucial to study alternative solutions that trade optimality for the
much needed computation speed.

**Newly emerging network functions.** There is no guarantee that our results would hold for the newly emerging network functions: *i.e.*, last-level cache isolation may not be sufficient for performance isolation. Most of the functions we looked at are legacy network functions that are implemented in software. The flexibility that software provides, we hope, will encourage a wide range of applications which may exhibit a wildly different behavior (*e.g.*, aggressive on the memory controller). Therefore more QoS knobs on other processor-scoped shared resources or even alternative method of performance isolation may be necessary.
Bibliography


[38] Intel® data plane development kit (dpdk), 2015.


