Building Evolvable Networks: Flexible and Predictable Packet Processing

THÈSE N^{O} 6721 (2015)

PRÉSENTÉE LE 28 AOÛT 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DES RÉSEAUX
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Mihai DOBRESCU

acceptée sur proposition du jury:

Prof. J.-P. Hubaux, président du jury
Prof. A. Argyraki, Prof. W. Zwaenepoel, directeurs de thèse
Prof. S. Ratnasamy, rapporteuse
Prof. G. Pierre, rapporteur
Prof. R. Guerraoui, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

I've missed more than 9000 shots in my career.
I've lost almost 300 games.
26 times, I've been trusted to take the game winning shot and missed.
I've failed over and over and over again in my life. And that is why I succeed.
—Michael Jordan

To my wonderful family...

Carmen & Ion, Ionut, Cristina

Acknowledgements

No man is an island entire of itself;
every man is a piece of the continent,
a part of the main.

John Donne

I am very grateful to all those I worked with, to those who advised me, and to those who simply stood by me during my PhD. Each one of these people influenced who I am, both professionally and personally, and my PhD experience would not have been the same without them.

First of all, I would like to thank Katerina Argyraki for being both a great advisor and a great person to work with. Katerina showed me what it takes to be an outstanding researcher and never let me aim for anything less. She was always there to guide me on how to pose the right questions, how to take a problem which seems impossible at first, break it down into pieces and start from there, or how to separate the fundamental from the incidental. Her dedication to research is inspiring and I am extremely grateful for all the time, energy and patience she has put into my development as a researcher. She has been the perfect combination between demanding and understanding, each at the right moment. I was lucky to have her as my advisor.

I am also grateful to the other professors who guided, advised and helped me throughout this time: Willy Zwaenepoel and George Candea for always finding the time to help and offer feedback, a piece of advice or an encouragement; Sylvia Ratnasamy, for the great experience I had while working with the Intel Research team.

My research was jointly done with Katerina Argyraki, Norbert Egi, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Eddie Kohler, Allan Knies, Maziar Manesh and Sylvia Ratnasamy. I appreciated the opportunity to work with these amazing people and I want to thank them for sharing their knowledge and enthusiasm with me. I am also grateful for the help and constructive feedback offered by my colleagues: Stefan Bucur, Michele Catasta, Vitaly Chipounov, Jiaqing Du, Ming Iu, Johannes Kinder, Nikola Knezevic, Vova Kuznetsov, Ovidiu Mara, Dimitri Melissovas, Alexandra Olteanu, Iris Safaka, Simon

Schubert, Radu Stoica, Jonas Wagner and Cristian Zamfir. Each of them was willing to share his time and energy to engage in productive discussions and provide insightful comments. I also want to thank the jury members who took the time to review this thesis and provided really valuable feedback.

EPFL provides a great environment for a PhD student, a place that is both very competitive and exciting. While this can be challenging at times, it also represents a good opportunity to meet and share ideas with the best people out there. Every day you get the chance to find out something interesting just by going for lunch or while waiting for the metro. I want to thank my colleagues in NAL, LABOS and DSLAB, who make the daily life at EPFL feel competitive and challenging, but also friendly and exciting. To my 5 years office-mate, Jiaqing, a special thank you—it was always good to know there is someone willing to bounce ideas or have a coffee with me. For our long discussions on academic life, economics or politics, for their advice and encouragements, many thanks to Dan, Dimitri, Duygu, Iris, Pavlos, Pinar, Michele, Ming, Radu, Simon.

Throughout these years, my family has been amazingly understanding and helpful. I want to thank my parents, Carmen and Ion, for raising me to be strong and independent and for encouraging me not to take the easy route in life, but always look for the one that inspires me the most. I am also grateful to my brother, Ionut, for always listening to me and trying to provide a good piece of advice based on a broad experience in the telecom industry. A special thanks to Cristina, who was my daily source of energy and optimism—thank you for standing beside me during all these years, for your patience and for always finding the strength to be supportive.

Lausanne, July 2015

Abstract

Software packet-processing platforms—network devices running on general-purpose servers—are emerging as a compelling alternative to the traditional high-end switches and routers running on specialized hardware. Their promise is to enable the fast deployment of new, sophisticated kinds of packet processing without the need to buy and deploy expensive new equipment. This would allow us to transform the current Internet into a programmable network, one that can evolve over time and provide a better service for the users.

In order to become a credible alternative to the hardware platforms, software packet processing needs to offer not just flexibility, but also a competitive level of performance and, equally important, predictability. Recent works have demonstrated high performance for software platforms, but this was shown only for simple, conventional workloads like packet forwarding and IP routing. And this was achieved for systems where all the processing cores handle the same type/amount of traffic and run identical code, a critical simplifying assumption. One challenge is to achieve high and predictable performance in the context of software platforms running a diverse set of applications and serving multiple clients with different needs. Another challenge is to offer such flexibility without the risk of disrupting the network by introducing bugs, unpredictable performance, or security vulnerabilities.

In this dissertation we focus on how to design software packet-processing systems so as to achieve both high performance and predictability, while maintaining the ease of programmability. First, we identify the main factors that affect packet-processing performance on a modern multicore server—cache misses, cache contention, load-balancing across processing cores—and show how to parallelize the functionality across the available cores in order to maximize throughput. Second, we analyze the way contention for shared resources—caches, memory controllers, buses—affects performance in a system that runs a diverse set of packet-processing applications. The key observation is that contention for the last-level cache represents the dominant contention factor and the performance drop suffered by a given application is mostly determined by the number

of cache references/second performed by the competing applications. We leverage this observation and we show that such a system is able to provide predictable performance in the face of resource contention. Third, we present the result of working iteratively on two tasks: designing a domain-specific verification tool for packet-processing software, while trying to identify a minimal set of restrictions that packet-processing software must satisfy in order to be verification-friendly. The main insight is that packet-processing software is a good fit for verification because it typically consists of distinct pieces of code that share limited mutable state and we can leverage this domain specificity to sidestep fundamental scalability challenges in software verification. We demonstrate that it is feasible to automatically prove useful properties of software dataplanes to ensure a smooth network operation. Based on our results, we conclude that we can design software network equipment that combines both flexibility and predictability.

Keywords: *Programmable Networks, Software Packet Processing, Predictability, Resource Contention, Scheduling, Verification, Symbolic Execution*

Résumé

Les plateformes logicielles traitant des paquets—dispositifs de réseau fonctionnant sur des serveurs d'usage général—émergent comme une alternative convaincante aux commutateurs réseau et routeurs traditionnels de haute gamme fonctionnant sur des équipements spécialisés. La promesse est de permettre la mise en place rapide de types de traitement nouveaux et sophistiqués sans avoir besoin d'acheter et de déployer des nouveaux équipements coûteux. Cela nous permettrait de transformer l'Internet actuel en un réseau programmable, un réseau qui peut évoluer au fil de temps et fournir un meilleur service aux utilisateurs.

Afin de devenir une alternative crédible aux plateformes hardware, les logiciels de traitement des paquets doivent offrir non seulement flexibilité, mais aussi un niveau compétitif de performance et, tout autant important, de prévisibilité. Des projets récents ont démontré une bonne performance pour les plateformes logicielles, mais cela n'a été démontré que pour les charges de travail simples et conventionnelles, comme la transmission des paquets et le routage IP. Ceci a été réalisé pour les systèmes où tous les cœurs du processeur traitent le même type/quantité de trafic et exécutent du code identique, une hypothèse simplificatrice critique. L'un des défis est d'atteindre une performance élevée et prévisible dans le cadre des plateformes logicielles exécutant un ensemble varié d'applications et servant plusieurs clients ayant des besoins différents. Un autre défi est d'offrir une telle flexibilité sans courir le risque de perturber le réseau en introduisant des bugs, une performance imprévisible, ou des vulnérabilités de sécurité.

Dans cette dissertation, nous nous concentrons sur la façon de concevoir des systèmes logiciels de traitement des paquets qui atteignent à la fois une bonne performance et prévisibilité, tout en conservant la facilité de programmation. Tout d'abord, nous identifions les principaux facteurs qui affectent la performance de traitement des paquets sur un serveur multicoeur moderne—défauts de cache, disputes de cache, équilibrage de charge à travers des cœurs du processeur—et nous montrons comment paralléliser la fonctionnalité à travers les cœurs disponibles pour maximiser le débit des paquets. Deuxièmement, nous analysons la façon dont la dispute pour des ressources partagées—

caches, contrôleurs de mémoire, bus—affecte la performance d'un système qui exécute un ensemble varié d'applications de traitement des paquets. L'observation clé est que la dispute pour le dernier niveau de cache représente le facteur dominant de dispute et la baisse de performance subie par une application donnée est principalement déterminée par le nombre de références de cache/seconde effectuées par les applications concurrentes. Nous profitons de cette observation et nous montrons qu'un tel système est capable de fournir des performances prévisibles vis-à-vis de la dispute pour des ressources. Troisièmement, nous présentons le résultat d'un travail itératif sur deux tâches : la conception d'un outil de vérification spécifique à un domaine pour des logiciels de traitement des paquets, tout en essayant d'identifier un ensemble minimal de restrictions que les logiciels de traitement des paquets doivent satisfaire pour être vérifiables. L'intuition principale est que les logiciels de traitement des paquets sont adéquats pour vérification, car ils sont composés généralement de pièces distinctes de code qui partagent un état modifiable limité et nous pouvons profiter de la spécificité du domaine pour éviter des défis fondamentaux d'évolutivité dans la vérification des logiciels. Nous démontrons qu'il est possible de prouver automatiquement des propriétés utiles des dataplanes logicielles pour assurer un bon fonctionnement du réseau. À la suite de nos résultats, nous concluons qu'on peut concevoir un équipement de réseau logiciel qui combine à la fois flexibilité et prévisibilité.

Mots-clés *Réseaux Programmables, Logiciels de Traitement des Paquets, Prévisibilité, Disputes des Ressources, Planification, Vérification, Exécution Symbolique*

Contents

List of Figures

List of Tables

1 Introduction

The last thing one discovers in composing a work is what to put first.

Blaise Pascal

1.1 Flexible Packet Processing

Traditionally, the development of network equipment (switches, routers, middleboxes) has focused primarily on achieving high performance for a limited number of packet-processing applications. However, as major Internet service providers (ISPs) compete in offering new services to their customers (e.g., video-on-demand, mobility) and networks need to perform more sophisticated functionality (e.g., application acceleration, intrusion detection), there is an increasing need for network equipment that is programmable and extensible.

Having a network built out of programmable devices would allow the fast experimentation of new, sophisticated packet-processing applications without the need to buy and deploy expensive new equipment. This can set the ground for developing an evolvable Internet, where the functionality of the network is not tied to particular hardware vendors [9, 15, 43, 56]. This way, the network can better adapt to the evolving needs of its users by continuously updating the functionality of devices located at strategic points in the network [64].

Currently available network equipment offers either high performance or programmability. On the one hand, there is the high-end network equipment that offers very good performance, but for a limited, predefined set of processing functions. These are typically built using specialized hardware (e.g., application-specific integrated circuits, custom network processors) and rely on closed software, which makes it very difficult, if not impossible, to program or extend their functionality. On the other hand, there is the

software network equipment that is built using general-purpose, off-the-shelf servers, and where all the significant packet-processing steps are implemented in software. These are fairly easy to program using a familiar programming environment and operating system, similar to the way we program end-systems today, but they are considered to be suitable only for small enterprises that require low packet-processing rates [6].

The challenge is to build network infrastructure that is programmable, but also capable of high performance and, equally important, predictable. This can be tackled starting from one of two extreme points: i) one approach is to start from the existing high-end, specialized devices and retro-fit programmability into them; ii) another approach is to start from software platforms running on commodity hardware and optimize their packet processing to achieve a competitive level of performance and predictability.

In this dissertation, we tackle the problem from the software end of the design spectrum, and we explore the feasibility of high-performance, predictable networking platforms that are built using commodity servers. Software packet-processing platforms offer several advantages compared to traditional hardware platforms: a familiar programming environment and all the advantages inherited from the PC-based ecosystem, such as low costs due to large-volume manufacturing, a widespread support chain and fast developments in platform and power management. At the same time, software platforms promise to enable the development of *truly* programmable networks, where changing the network's functionality would be equivalent to performing a simple software upgrade.

For a long time, the common perception was that the general-purpose processors are not capable of high-speed packet forwarding, leading to the entire development of specialized equipment. However, the recent advances in server technology are changing the scene: inside one server we now have multiple multicore processors interconnected with high-speed point-to-point links, integrated DRAM controllers, high bandwidth PCIe buses for the I/O transfer and multi-port multi-queue network cards (NICs). All these features enable high-speed parallel processing, which is well-suited for packet processing. As a result, in recent years, we have seen emerge both from industry and the research community several projects that leverage the increased level of parallelism present in the modern hardware and demonstrate that software packet processing can deliver good performance. From the commercial world we have seen network appliances running on general-purpose processors that are capable of handling tens of Gbps [8]. From the research community, we have seen a software router architecture, that parallelizes the functionality both across multiple cores inside a server and across multiple servers inside a cluster and can scale to hundreds of Gbps [34]. More recently, we have seen several projects that propose I/O optimized software architectures and improve the processing capability of one server to deliver line rates of 10s of Gbps [43, 44, 56, 57, 65]. All these programmable platforms are running on general-purpose hardware and deliver a competitive level of performance that scales well beyond the requirements of a small enterprise network.

1.2 Predictable Packet Processing

One aspect that has received less attention from the community is the reliability of a software packet-processing platform. A system of non-trivial size that is subject to frequent updates is typically affected by performance and behavior bugs, as well as security vulnerabilities. Network operators are unlikely to accept the risk that an unlucky configuration could result in an unpredictable drop in network performance; or that a specially crafted packet, potentially sent by a malicious end-host, could disable the packet-processing pipeline and disrupt network operation. These situations could lead to violations of service-level agreements and customer dissatisfaction, which can have negative implications for business. It makes sense that network operators are skeptical about the vision of software platforms that are frequently reprogrammed in response to the needs of the users, as they were skeptical a decade ago about Active Networks [71,72]. As anecdotal evidence, when we discussed previous results on software packet processing with hardware-device manufacturers, their main concern was not performance, but reliability: "If we allow third-party code to be added into the dataplane, there will be functionality bugs and there will be performance bugs. Who will pay for the customer support? What is the extra price that we need to pay in order to get programmability?" This concern is justified, given the history of transitioning from hardware to software implementations in other domains like consumer electronics, car industry, and aviation industry, where software has dramatically increased the speed with which new features are implemented, but has also created the perception that it led to products that are more fragile and more likely to misbehave [10,12,28].

In recent years we have seen several projects that demonstrate software platforms capable of high performance [34,43,44,56,57,65], but this was shown only for simple, *conventional* workloads like packet forwarding and IP routing. Moreover, these results have been achieved under the simplifying assumption of *uniformity*: all the processing cores handle the same workload—they see the same amount of traffic and they perform exactly the same kind of packet processing. This way, the system can be optimized for a particular workload, and any minor deviation from this setup could result in a significant performance drop. For instance, in the context of the RouteBricks prototype [34], a general-purpose, eight-core Nehalem server was able to perform IP routing at 24Gbps, but only after carefully tuning the entire system (e.g., manually setting buffer sizes and batch parameters).

We need to address several challenges in order to make software platforms a credible alternative to specialized hardware. First, we need to demonstrate that software platforms can deliver high performance for a range of sophisticated packet-processing applications, while maintaining *ease of programmability*. The promise offered by software packet-processing platforms is that they would enable the network to evolve *beyond* conventional IP routing. Therefore, it is not enough to demonstrate that a software platform can achieve good performance in a very particular context.

Second, we need to demonstrate that software platforms can deliver high and *predictable performance* for a diverse set of packet-processing applications, while serving multiple clients each with different needs. A software device is expected to run several packet-processing applications, each applied to a different subset of the incoming traffic. For instance, it has been shown that we can significantly reduce the networking costs by dynamically allocating packet-processing tasks to network devices and, this way, cutting down the network provisioning by half [66]. At the same time, having a system that needs to support an evolvable set of applications, possibly developed by different entities, creates the fear of unpredictable performance. That is because various software processes share the same software/hardware infrastructure and they interact with each other in unpredictable ways that can lead to a potentially significant drop in performance: false sharing [21], unnecessarily shared data structures [23], contention for shared hardware resources like caches, memory controllers and buses [70].

Third, we need to be able to prove useful properties of software platforms so as to ensure smooth network operation. The flexibility offered by a software platform brings along the threat of software bugs that lead to unpredictable behavior and performance. Consider a piece of packet-processing code that enters an infinite loop or results in a kernel panic upon receiving a particular type of packet. If we allow this piece of code to run inside a networking device, any attacker could disrupt the network operation by forcing the device to stop processing packets. We need to detect these situations before we deploy the code inside the network. At the very least, we need to be able to prove properties that, in the case of hardware devices, are either taken for granted or can be proved using practical techniques [46–48, 58, 74]: *crash-freedom*, which means that no packet sequence can cause the dataplane to stop executing; *bounded-execution*, which means that no packet sequence can cause the execution of more than a known, reasonable number of instructions; or *filtering* properties—for instance, "any packet with source IP A and destination IP B will be dropped" if we instruct the network device to do so.

In this dissertation we show that it is feasible to build software packet-processing systems that achieve both high performance and predictability, while maintaining the ease of programmability. Hence, we argue that software platforms represent a viable approach to building networks that are flexible and extensible and they could represent the basis of an evolvable Internet.

1.3 Contributions

In this dissertation we make the following contributions:

- First, we show how to parallelize the functionality of a software packet-processing system across the processing cores of a state-of-the-art, general-purpose server in order to maximize throughput (Chapter 3).

  In particular, we analyze the two available parallelization approaches: (a) "cloning", where each core handles all the processing steps for a subset of the incoming traffic; (b) "pipelining", where each core handles a subset of the processing steps for all incoming traffic. We identify three key factors that determine which of these two approaches works better (inter-core synchronization, cache contention, and load-balancing of data-flow functionality), and we design a simple optimization process that automates the parallelization decision.

  While each of the two parallelizing approaches could lead to higher throughput depending on the server resources and particular application requirements, we show that, for most practical scenarios involving realistic workloads, the cloning strategy results in higher throughput.

- Second, we present a software packet-processing system that provides predictable performance in the face of resource contention while running a diverse set of applications and serving multiple clients (Chapter 4).

  In particular, we analyze the way contention for shared resources—caches, memory controllers, buses—affects performance and we make the following key observations:

  - In the case of software packet-processing systems, the dominant contention factor is the contention for the last-level cache.
  - The performance drop suffered by a given application is mostly determined by the number of last-level cache references/second performed by the competing applications.

  By relying on these observations, we predict, for each application running in our system, the performance drop suffered due to contention with other applications sharing the same hardware platform, with an error smaller than 3%.

- Third, we show that it is feasible to automatically prove useful properties of software packet-processing systems to ensure a smooth network operation (Chapter 5).

  In particular, we present a verification tool that takes as input a software dataplane and proves that it does or does not satisfy properties like crash-freedom, bounded-execution, and filtering. The key insight is that packet-processing software can be written in a verification-friendly way because it typically consists of distinct pieces of code that share limited mutable state.

By adapting existing verification techniques and leveraging the domain specificity of software dataplanes, we perform complete and sound verification of stateless and simple stateful dataplanes within tens of minutes. A naïve application of existing verification techniques fails to complete the same tasks.

We conclude that it is feasible to design a high-performance software packet-processing system that combines both *flexibility* and *predictability*.

1.4 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we provide the necessary background for this dissertation: we provide a brief introduction to Click [52], the programming model that we adopt in our work; we present what is a good way to configure a software packet-processing system in order to optimize performance; and what is the performance range we can expect from such a system. In Chapter 3 we show how to parallelize the functionality of a software packet-processing system across the available resources in order to maximize the throughput. In Chapter 4 we analyze how contention for shared resources affects performance in a system that runs a diverse set of applications, each with different requirements and we show a system that provides predictable performance notwithstanding the shared hardware resources and the unavoidable contention. In Chapter 5 we discuss what are the particularities of a software packet-processing system that make it verification-friendly, and we present a tool that proves useful properties of software dataplanes. In Chapter 6 we conclude the dissertation by summarizing our findings.

2 Background: Software Packet Processing

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton

In this chapter, we first present the most successful approach to achieving flexibility in software packet processing (Section 2.1), then we present the most successful approach to achieving high performance and the main results (Section 2.2).

2.1 Flexibility

In this dissertation, we focus on pipelines developed with Click [29, 52], a programming model and a software architecture used for building modular packet-processing platforms (for brevity also referred to as "routers") that can be easily configured and extended. To the best of our knowledge, this is the most successful research effort that sought to combine flexibility with performance in a software packet-processing system.

The Click programming model is based on two types of entities: *elements* and *connections*. A router configuration is represented using a directed graph with elements as vertices and connections as edges. Packets flow from one element to another along the edges of the graph:

- *Elements* represent packet-processing modules that specify the code being executed whenever a packet passes through that particular element. A typical Click element handles simple packet-processing tasks like packet classification, longest prefix matching, interaction with network devices or packet queuing. Each element has a number of input ports and a number of output ports, which serve for transferring packets between elements.

- *Connections* represent links between one output port of one element and one input port of another element. They define the possible paths through which packets may flow in the graph.

A Click configuration uses the Click declarative data-flow language to specify the high-level behavior of the packet-processing system. We illustrate this using the simple processing flow in Figure 2.1:

- The *PollDevice* element is responsible for network device interaction, the receiving part. It reads a packet (or multiple packets) from the input interface (*eth*0, *eth*1, in this case), and it passes the packet to the next processing element in the pipeline (*ProcessingElement*1, *ProcessingElement*2, in this case).
- The *ProcessingElement*1 and the *ProcessingElement*2 elements apply specific packet processing on the packets that they observe (e.g., packet filtering, network address translation, etc.), then they place the packets into the *Queue*.
- The *ToDevice* element is responsible for network device interaction, the sending part. It takes packets from the *Queue* and sends the traffic out on the output interface *eth*2.

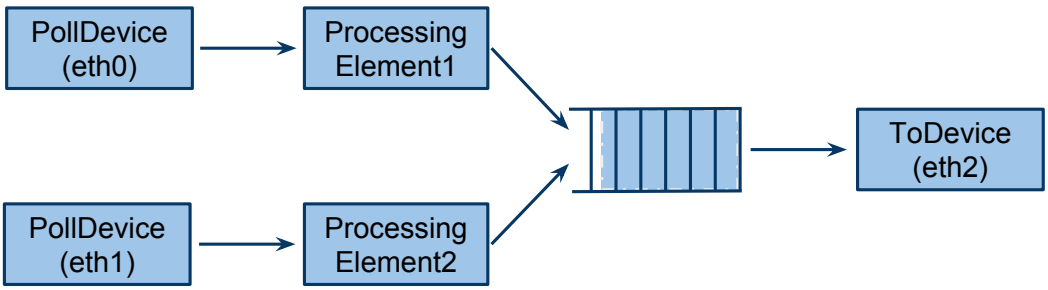

Figure 2.1: A simple Click configuration.

To create a Click configuration, one can use one of the many packet-processing elements that come with the Click distribution and/or implement their own elements. The Click distribution provides a comprehensive set of packet-processing elements, for instance, for packet classification, packet storage, traffic shaping, processing of standard protocols like Ethernet, ARP, IP, UDP, TCP, etc. To implement a new packet-processing element, one must extend the abstract class *Element* provided by the Click distribution. This typically requires implementing just a few functions: *class_name*, *configure*, *initialize*, *cleanup*, *port_count*, *simple_action*, etc. In order to implement custom packet processing, an element will need to override one of the functions that are called as the router processes packets (e.g., *simple_action*, *pull*, *push*). A typical element will receive a packet from

the upstream element, will apply the custom processing implemented by one of these functions and then forward the packet to next downstream element. For instance, the *simple_action* function (*Packet* $*$ *simple_action*(*Packet* $*$ *p*)) takes as input a packet (a pointer to a *Packet* object that consists of a data buffer and a set of annotations like header pointers, timestamp, custom annotations, etc.), and outputs the transformed packet (or *null*, if the packet is dropped).

2.2 Performance

Historically, the Achilles' heel of software packet-processing platforms has been performance. Several years ago, when hardware platforms were able to forward traffic at aggregate rates beyond the 1Tbps mark [1], software platforms had trouble scaling beyond the 1–5Gbps range [7]. However, in recent years we have seen several efforts that sought to improve performance and, in this section, we will summarize the main challenges and how they are addressed in Click [29, 52] and RouteBricks [34], two key projects relevant for this dissertation.

The recent advances in server technology (e.g., multiple multi-core CPUs, multi-queue network cards, multiple memory controllers) enable high-speed parallel processing, which is particularly suitable for packet-processing workloads. Figure 2.2 shows a high-level description of a state-of-the-art general-purpose server: it has multiple sockets/processors (CPUs) that are connected with each other using a mesh of high-speed point-to-point links (e.g., Intel QuickPath Interconnect—QPI, AMD Hypertransport); CPUs access the main memory through integrated memory controllers with multiple memory channels, which leads to high aggregate memory bandwidth; each CPU contains multiple processing cores and uses a 3-level cache hierarchy to reduce memory access time—typically, each core has private Level 1 (*L*1) and Level 2 (*L*2) caches, whereas the Level 3 (*L*3) cache is shared among all the cores of the same CPU; CPUs access the I/O subsystem through an I/O Hub (IOH) that connects several network interface cards (NICs) using high-speed PCIe buses; NICs contain multiple hardware queues, which allows multiple cores to access the device in parallel.

Before we go into details on how to achieve high performance, we first provide a brief overview of a packet's life cycle inside a software packet-processing system. This typically involves the following operations: for each packet that is processed, the host needs to allocate a buffer that stores data and a socket buffer descriptor that stores metadata (e.g., buffer length, pointer to MAC header, pointer to IP header); the host is also responsible to create and initialize a NIC descriptor ring for each network queue; each descriptor contains a pointer to a buffer where the data is supposed to be copied to/from; when a new packet is received, the NIC uses the direct memory access (DMA) engine to fetch the next available descriptor and transfers the data into the corresponding buffer; once the transfer is complete, the DMA updates the corresponding descriptor and raises an

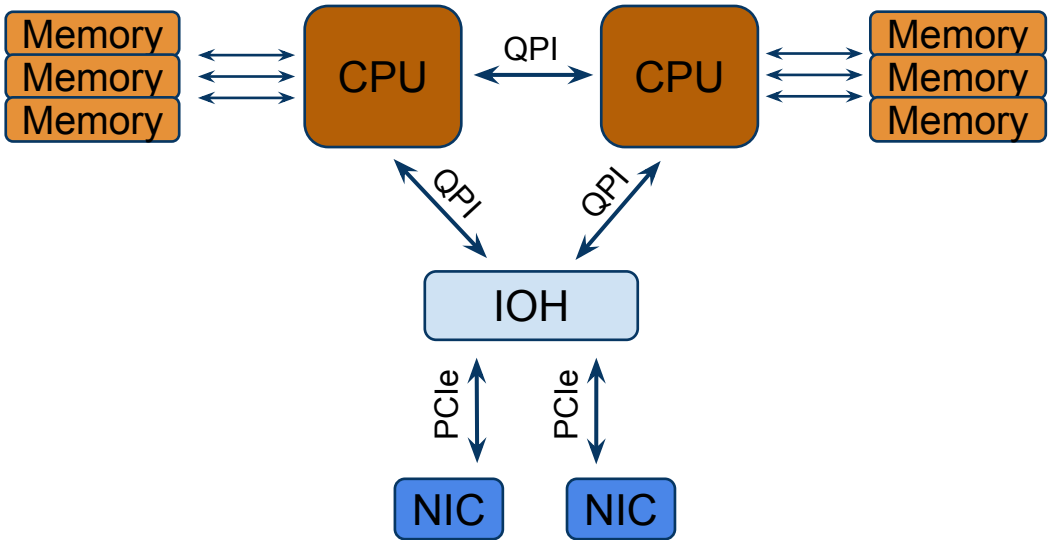

Figure 2.2: Hardware architecture for a general-purpose server.

interrupt to notify the host; as a note, in order to avoid the interrupt overhead [61] and the synchronization between threads and interrupts, Click device drivers disable interrupts and use polling to check for new packets; the CPU reads the packet headers and, potentialy, packet payload and, depending on the type of processing implemented, it perfroms extra actions—e.g., determine the output port, encrypt the packet payload, etc.; once the processing is complete, the CPU updates the transmit ring descriptor and notifies the NIC that a new packet is ready for transmission; the NIC takes the packet from the main memory using DMA and sends it out on the wire.

Software packet-processing platforms running on general-purpose hardware are capable of high performance. For instance, a single RouteBricks server, build on top of Click, forwards up to 19 million packets per second (Mpps) and achieves traffic rates of up to 24.6Gbps [34]. This performance is achieved on a dual-socket Intel Nehalem platform: each CPU has 4 cores running at 2.8GHz and a shared L3 cache of 8MB; the server contains two dual-port 10Gbps NICs installed on PCIe1.1 x8 slots. While the performance limiting factor is determined by the type of packet processing involved and the characteristics of the input traffic (e.g., packet size), it is worth mentioning that for "simple forwarding" or "IP routing" workloads, the server is CPU-bottlenecked when forwarding small-size packets, and I/O-bottlenecked when forwarding realistic-size packets; for "IPsec packet encryption", the server is CPU-bottlenecked, irrespective of the size of the input packets.

Achieving good performance in a software packet-processing system requires paying attention to low-level details in order to leverage parallel execution and ensure good scalability. For instance, special care is required when handling interaction with the network device, memory management or task scheduling. We now describe several

strategies that are employed by Click [29, 52] and Routebricks [34] so as to achieve high performance. While the reader might be familiar with these techniques and consider them as mostly known, common-sense practices when doing parallel programming, our aim is to emphasize that details matter when designing a system that needs to forward millions of packets per second, i.e., one can not just take a server out of the box and start forwarding traffic at 10s of Gbps.

Memory Management

Processing one packet requires at least two memory allocation/freeing operations: one for the socket buffer descriptor and one for the data buffer itself. A naïve system uses a single pre-allocated memory pool that is shared by all cores. This requires locking for each operation and makes memory management an expensive task in terms of per-packet CPU cycles. Moreover, a single memory pool makes it more likely to have one buffer freed by one core and allocated by another, which results in extra cache misses and increases the per-packet CPU cycles. Instead, Click takes a different approach and maintains *a separate memory pool for each core*: each core frees memory only into its own pool; whenever a core needs more buffers, it first tries to allocate from its own pool, and, only if that fails, it allocates from another core's pool or from the systems' general-memory pool. In this case, memory freeing is a lock-free operation, and for memory allocation the cost of locking can be as low as the cost of a read operation from $L1/L2$ cache, if allocating from the local memory pool.

Shared Data Structures

Using locks to synchronize accesses to mutable data structures affects performance in two ways: first, execution is stalled whenever there is contention; second, having multiple CPUs accessing the same lock results in compulsory cache misses that require expensive off-chip data transfers. Every mutable data structure should be touched by as few CPUs as possible in order to avoid synchronization overhead. Therefore, it is recommended to replicate state and *maintain per-CPU data structures* for each element, whenever the logic of the application allows it. For instance, if packets of the same TCP flow are always processed by the same CPU, an element can maintain state using a per-flow data structure and avoid locking. If state replication is not possible, efficient techniques for synchronization should be used: atomic hardware operations for data structures that are rarely updated; read-write locks for data structures that are mostly read and rarely written; spin locks implemented with atomic *compare&swap* instructions, in all the other situations.

Device Interaction

If a receive or a transmit queue of a network device is accessed by multiple cores, each core must lock the queue before accessing it, which is an expensive operation in the case of multi-Gbps interfaces. Therefore, the best practice is to *use one core per queue*: the multi-queue NICs allow one to use as many queues as cores in the system, thus, each core can access the device using its own dedicated queue, without having to ever synchronize with other cores.

Batch Processing

Packet processing involves several per-packet overheads—for instance, "book-keeping" operations like reading/updating the socket buffer descriptors and the ring descriptors. The cost of these overheads can be amortizing over multiple packets if *packets are processed in batches*. For example, when the *PollDevice* element fetches packets from the network device, it does so in batches to amortize the book-keeping costs and allow the driver to prefetch data into the cache. Similarly, the *ToDevice* uses batching to dequeue packets from an upstream *Queue* (to amortize the cost of accessing the queue) or to notify the transmitting device (to amortize the cost of notification). Batching also helps when allocating/freeing packet buffers because the cost of manipulating the memory pool can be amortized over multiple packets.

Task Scheduling

Click creates a separate kernel thread for each core in the system. Each thread maintains a private work list of schedulable elements (e.g., *PollDevice* element, *ToDevice* element) that are scheduled periodically in a round-robin fashion. Whenever an element gets the control of the CPU, it will not be interrupted until the packets are either placed in a *Queue* element, sent out in the network or dropped. While we discuss in detail the problem of element-to-core allocation in Chapter 3, we note that the main challenge for a good scheduling strategy is to distribute the elements to cores in a way that makes good use of the available resources (e.g., load balance the workload evenly across the processing cores, utilize the available cache space efficiently). Intuitively, a naïve approach that distributes an equal number of elements to each work list is not efficient, since different elements may have different complexities, which would lead to having idle CPU cores.

3 Parallel Execution

The greatest improvement in the productive powers of labour, and the greatest part of skill, dexterity, and judgment with which it is any where directed, or applied, seem to have been the effects of the division of labour.

Adam Smith

3.1 Introduction

Software packet-processing platforms represent an appealing alternative to the traditional hardware platforms. They promise to enable the development of truly programmable networks, where changing the network's functionality is equivalent to performing a simple software upgrade. This would allow us to try out new kinds of packet processing without the need to buy and deploy expensive new equipment.

The recent advances in modern server technology have demonstrated that software packet-processing platforms are capable of high performance *provided they are carefully programmed*. For instance, in the context of the RouteBricks prototype, a general-purpose server equipped with two quad-core Nehalem processors was able to perform IPv4 routing at 24Gbps [34]. However, this was achieved only after tedious manual tuning. Moreover, all packets were subjected to the same kind of packet processing (IP routing) and all the necessary data structures (the forwarding table) were able to fit in the cache. A minor deviation from this careful setup could result in a significant performance drop.

Yet the allure of software packet processing is that it could enable the network to evolve *beyond* conventional IP routing. To prove this, it is not enough to demonstrate that software platforms can achieve good performance in a very particular context. The fear

is that once we start deploying more sophisticated packet-processing applications, we will increase the complexity of the system to the extent that we will either face performance issues or we will lose the ease of programmability (i.e., a significant manual effort would be required to configure the system in order to achieve the desired level of performance). Therefore, we need to demonstrate that they can deliver high performance for a range of sophisticated packet-processing applications without losing programmability.

In this chapter we analyze how we can simultaneously achieve both *high performance* and *ease of programmability* in a software packet-processing platform. More concretely, our high-level goal is to automate the process of parallelizing a packet-processing flow. For this, we set to answer the following questions: given a particular multicore server architecture and a particular set of packet-processing applications, how do we parallelize the functionality across the available cores so as to maximize throughput? and what do we need to know about the server architecture and each application in order to perform such an optimization?

In some sense, our work builds on early work by Chen and Morris on SMP Click [29] that sought to extend Click for multi-processor architectures. They evaluated two approaches to scheduling a Click processing pipeline across multiple cores:

- the "dynamic" approach, where processing elements are assigned to cores in a dynamic manner that seeks to balance CPU utilization across cores. In this approach, the element-to-core assignment doesn't take into account the overheads that result from partitioning the processing of a packet across different processors (e.g. compulsory cache misses);

- the "static" approach, which relies on a static assignment of elements to cores. In this approach, the element-to-core assignment is determined manually by an "ambitious" programmer and is fixed for the duration of the program's execution.

Their results suggest that the static approach typically outperforms the dynamic one. Our goal is to automate the manual assignment decision made by the ambitious programmer of SMP Click and, subsequently, to allow this scheduling decision to be made in a dynamic manner at runtime.

The rest of the chapter is organized as follows: in Section 3.2, we describe what are the general approaches for exposing parallelism in a multicore software platform and we provide the insights motivating why each approach might give better performance in different scenarios. In Section 3.3, we describe the overheads incurred by each approach and formulate an optimization problem that determines which approach is better for a particular server architecture and a given workload. We summarize our findings in Section 3.4.

3.2 Exploiting Parallelism

At a high level, our goal is: given a packet-processing application, we want a method that automates the parallelization of the application so as to maximize its performance. This question falls, of course, under the general question of whether/how we can automatically parallelize code to exploit multicore hardware—a hard problem in a general context and the subject of much ongoing research on multicore systems. For tractability, we narrow our exploration to a specific data-flow programming model and consider the question of how to parallelize a single data-flow under several simplifying assumptions that we elaborate on as we introduce them in the text.

3.2.1 Programming Model

We want to determine how to configure a software packet-processing platform in order to combine both programmability and performance. Hence, we assume that our platform is programmed using the Click programming model [52]—to our knowledge, the most successful research effort that sought to combine these two properties. More specifically, we assume that a packet-processing application is written as a set of modular "elements", each specifying a packet-processing task. This way, the functionality of the application can be described as a data-flow graph, which specifies the sequence of elements traversed by each received packet. A simple example of a data-flow graph is shown in Figure 3.1. It consists of two elements (IP routing and NetFlow) and all the packets received by the platform are subjected first to IP routing, then NetFlow processing, before being forwarded to the next hop.

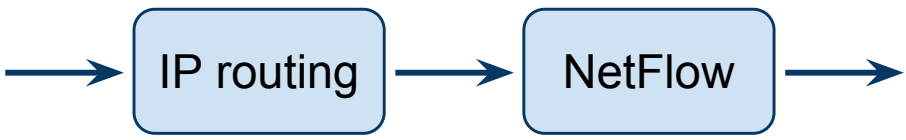

Figure 3.1: A simple data-flow graph that consists of two packet-processing elements: IP routing and NetFlow.

3.2.2 Parallelization Opportunities

When configuring a multicore software platform, a key question is how to parallelize the functionality across the available processing cores. We need to determine which part of the data-flow graph will be executed by which core. The "parallelization" strategy has a dramatic impact on the platform's performance—it is easy to imagine how overestimating or underestimating the number of cores required for a particular task can lead to a bad utilization of the hardware resources.

There are two general approaches to performing the parallelization (illustrated in Figures 3.2 and 3.3) [29].

- *Cloning*: in this approach, each core executes the entire data-flow graph. The incoming traffic is split across all cores and each core performs all the processing required for the corresponding subset of the incoming traffic. In this case, each packet is processed by only one core.
- *Pipelining*: in this approach, each core executes only a part of the data-flow graph. The data-flow graph is partitioned across multiple cores and each core performs only the corresponding part of the processing required for a given packet. In this case, each packet is processed by multiple cores.

Recent work that uses multicore general-purpose hardware for packet processing follows the cloning approach, because it has been shown to yield higher performance for simple workloads (e.g., where the router performs IP routing on all packets [34, 43]). But is this always the right choice?

Consider, for instance, the example in Figure 3.1, where we want to run IP routing and NetFlow (a widely deployed router application that collects per-flow statistics [2]) on a server architecture with two cores. Suppose each core has an αMB last-level cache (LLC), while each application uses a data structure (e.g., a forwarding table and a flow table, respectively) of approximately αMB. If we apply the cloning approach, each core runs both applications, which means that it accesses 2αMB of data through the αMB cache. On the other hand, if we apply the pipelining approach, each core runs only one of the two applications, which means that it accesses αMB of data through the αMB cache. Hence, each application accesses its entire data structure from the cache. In this particular example, it is possible that the pipelining approach leads to fewer cache misses and, fewer cache misses could translate to higher performance, depending on the server's bottleneck.

This simple, yet realistic scenario suggests that each approach—cloning or pipelining—may be the right choice to yield higher performance depending on the scenario. Extending this choice to multiple elements in a more complex data-flow results in a large space of potential parallelizations— cloning, pipelining and various combinations of the two (i.e., where some portions of the data-flow are cloned and others are pipelined).

3.2.3 Decision Process Overview

We want to answer the following question: given a multicore server architecture and a data-flow graph representing a set of packet-processing applications, which part of the graph should each core execute so as to maximize the system's performance?

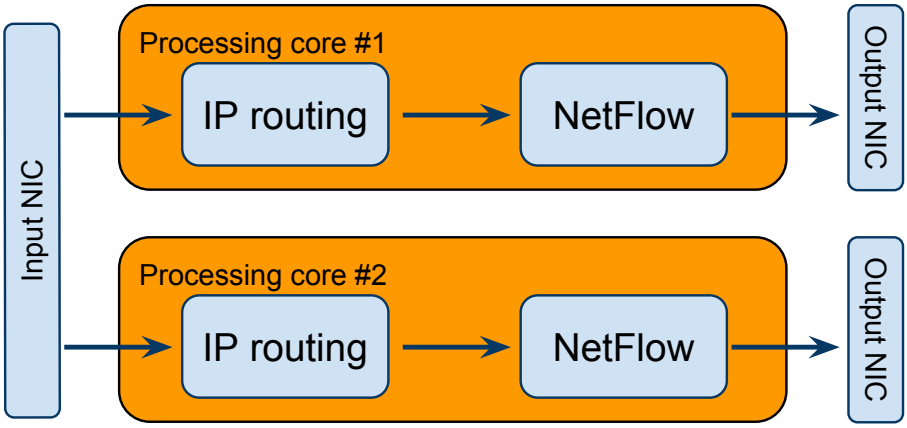

Figure 3.2: The cloning approach: incoming traffic is split among multiple cores and each core performs all processing for all packets it receives.

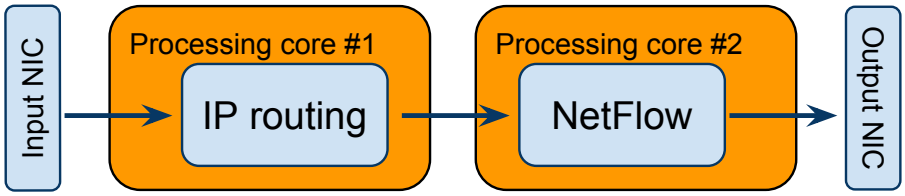

Figure 3.3: The pipelining approach: incoming packets traverse multiple cores and each core performs a different kind of packet processing.

We set out to design a decision process that takes as input (1) a server hardware profile; (2) a data-flow graph and a profile of each element in the data-flow graph; and outputs which element(s) should be run by each core. The server hardware profile should specify the set of available resources (e.g., number of processors, number of cores, cache sizes, etc.); the element profiles should specify the resource requirements for each element; the decision process should compare all the possible parallelization options (cloning, pipelining, or any combination of the two) and choose the option that maximizes the system's performance.

While there are multiple questions we need to answer in order for this strategy to work (e.g., what metrics should characterize a server profile or an element profile?), in this chapter we focus on the decision process and, in particular, on understanding the trade-offs between the cloning and pipelining approaches which is at the core of any decision process.

Hence, as a first step, we assume perfect knowledge of the server architecture and the applications running on it (i.e., the elements of the data-flow graph) and we want to

determine what factors we must consider in making the cloning-vs-pipelining decision. This boils down to understanding when and how the two approaches differ in the overheads they incur (e.g., cache misses, synchronization overheads) under different scenarios. Implicitly, understanding these overheads also reveals the extent of performance improvement that we stand to gain by making the "right" decision.

3.3 Cloning vs. Pipelining

In this section we identify and we analyze the overheads associated with different parallelization options. Note that, since our goal is to make the cloning-vs-pipelining decision, we only need to focus on the *distinguishing* overheads, those that are incurred under cloning but not pipelining, or vice versa.

We identify three distinguishing overheads:

- *synchronization* overhead;
- *cache contention* overhead;
- *pipeline imbalance* overhead.

We describe each of these overheads, we present experiments that illustrate the performance impact of each of them and formulate a simple optimization problem that seeks to weigh the competing overheads in order to maximize the overall packet-processing rate.

3.3.1 Experimental Setup

We use a general-purpose server, equipped with two quad-core Intel Nehalem Xeon 5560 processors; the four cores of each processor share an 8MB L3 cache, while each core has a private 256KB L2 cache and a private 32KB L1 cache. The server is also equipped with two network cards, each with two 10Gbps ports. Traffic arrives at all 4 ports at the same rate and all the traffic that is received at one port is sent out on the other port of the same card. The server runs SMP Click [29].

We use a data-flow graph with two processing elements, similar to the one in Figure 3.1. The only difference is that instead of IP forwarding and NetFlow, we use synthetic packet-processing applications. Each packet is processed by both processing elements. For each packet that is processed, element i (where $i \in \{1, 2\}$) reads the packet headers, then reads N_i random memory locations from an array of size S_i bytes, where N_i and S_i are configurable parameters. As we will see, although simple, this functionality captures those aspects of packet-processing applications that determine whether cloning or pipelining is the right choice for parallelization.

When we use the cloning approach, each of the 8 cores receives $\frac{1}{8}$ of the incoming traffic, executes the functionality of both processing elements, then sends the traffic out in the network. When we use the pipelining approach, each of the 4 cores of processor 1 receives $\frac{1}{4}$ of the incoming traffic, executes the functionality of the first processing element, then passes the traffic to the corresponding core of the processor 2 (i.e., core 1 on processor 1 passes the packets to core 1 on processor 2, core 2 on processor 1 passes the packets to core 2 on processor 2, etc.). The cores of processor 2 take the traffic, execute the functionality of the second processing element, then send the traffic out in the network.

In all the experiments that we present, the server is fed a workload of 64-byte packets, because this is the workload for which the server's performance is affected the most by the cloning-vs-pipelining decision. The reason for this is that the identified overheads affect performance only when the server is CPU bottlenecked (all overheads consist of extra per-packet cycles) and this happens for small-packet workloads.

In our setup, whether we use the cloning or pipelining approach, the server is CPU bottlenecked—the cores run out of processing cycles—and most cycles are spent waiting for memory accesses to complete, which makes sense because our processing elements do nothing but read data from memory.

3.3.2 Synchronization Overhead

With pipelining, each packet is processed by multiple cores that do not share a cache. This requires synchronization between multiple cores due to the following operations:

- *Metadata sharing.* In Click, one core "passes" a packet to another core by storing a pointer to the packet's socket buffer descriptor (the Linux-kernel data structure that contains the packet's metadata, e.g., where the packet is stored in memory) in a first-come-first-serve queue served by the second core. The enqueuing and dequeuing operations require extra cycles. Moreover, there is an extra cache miss when the second core reads the pointer. Finally, book-keeping the queue (e.g., when the cores check whether the queue is full/empty) leads to extra cache misses when the status of the queue has changed.

- *Content sharing.* Every time a core processes a new packet, it reads the contents of the corresponding socket buffer descriptor and the packet headers. Moreover, if the core performs deep-packet inspection, it also reads the contents of the packet. Each of these read operations results in compulsory cache misses when the socket buffer descriptor and the packet do not reside in a cache accessible by the core.

  With cloning, each packet is processed by one core, hence these read operations are performed only once. In contrast, with pipelining, these read operations are performed every time a packet is passed from one core to another, resulting in

extra cache misses and lower performance.

- *Memory recycling.* In Click, each core that reads a packet from the network (the "receiving" core) stores it in a pre-allocated memory pool, and there is a separate pool for each core.

  With cloning, the receiving core is the one that also sends the packet out into the network (the "transmitting" core); hence, once the packet is transmitted, the core can easily recycle the corresponding packet buffer back into its pool. In contrast, with pipelining, the receiving core is different from the transmitting core; hence, the transmitting core has two options for recycling the packet buffer: into the receiving core's pool, which requires synchronization between the two cores for accessing the same pool; or, into the transmitting core's pool, which will cause the receiving core to exhaust its pool and re-allocate memory—a costly operation in terms of CPU cycles.

We identified these operations as the main sources of synchronization overhead by running the following experiment. We configured the two processing elements with parameter values $S_1 = S_2 = 2MB$ and $N_1 = N_2 = 1$ memory access per packet (i.e., for each received packet, each element reads one random memory location from a 2MB array). We picked the values of S_i such that both elements can easily fit their data structures in the L3 cache and, this way, avoid cache misses due to cache contention.

Given this setup, the system achieves 8.56Gbps throughput with cloning and 2.89Gbps with pipelining (i.e., with pipelining, throughput drops by 66%). To understand this drop, we profiled our router under each approach. We found that cloning results in about 4 L3 cache misses per packet (two associated with the socket-buffer descriptors and two with the packet headers), whereas pipelining results in 13–14 additional cache misses per packet (which correspond to the synchronization operations mentioned above).

We conclude that pipelining introduces significant synchronization overhead, hence we expect cloning to perform better in scenarios where compulsory cache misses are the dominant cost factor in terms of cycles spent per packet.

3.3.3 Cache Contention Overhead

With cloning, each core runs all elements, which means that all elements contend for the cache (at all levels). This can lead to cache contention, when the aggregate working set size of the elements does not fit in the LLC (in our setup, the L3 cache). In contrast, pipelining allows different elements to use different L3 caches, which should reduce cache contention.

To understand the impact of cache contention on system performance, we repeated the last experiment multiple times, for different S_i and N_i values, and measured the "benefit"

of pipelining over cloning, i.e., the system's throughput under pipelining normalized by the system's throughput under cloning. Hence, a benefit above 1 means that pipelining performs better than cloning, whereas a benefit below 1 means the opposite.

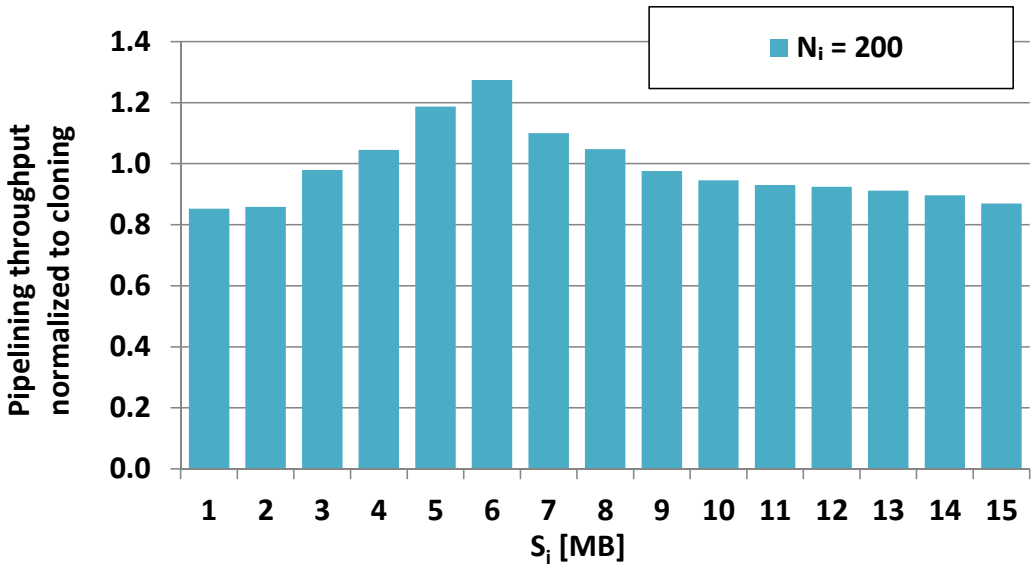

Figure 3.4: Cache contention overhead for $N_i = 200$

Figure 3.4 shows the benefit of pipelining over cloning for $N_i = 200$ memory accesses per packet, as a function of S_i. We see that the benefit of pipelining increases with S_i until it reaches a maximum point at $S_i = 6$MB, then it decreases. We explain this as follows:

1. For small numbers of S_i, the arrays of both elements fit in the L3 cache, hence neither cloning nor pipelining incur cache contention. Therefore cloning performs better because of the synchronization overhead of pipelining.

2. Beyond some point ($S_i = 3$MB), the array of one element fits in one L3 cache, whereas the arrays of two elements do not fit. Hence, pipelining performs better because cloning incurs cache contention whereas pipelining does not.

3. However, beyond a second point ($S_i = 6$MB), even the array of one element does not fit in the L3 cache. Hence, pipelining cannot avoid cache contention either, which causes its benefit to decrease.

4. Finally, beyond a third point ($S_i = 9$MB), cache contention under pipelining becomes so strong that the synchronization overhead of pipelining outweighs its cache-contention benefit over cloning, and cloning starts performing better again.

Figure 3.5 shows the benefit of pipelining over cloning for $N_i = 20$ memory accesses per packet, as a function of S_i. We see that cloning always performs better (the benefit of

pipelining over cloning is below 1) independently from S_i. This is because, when memory accesses are relatively infrequent, the L3 cache manages to absorb them to some extent, such that the cache-contention benefit of pipelining is outweighed by its synchronization overhead. We argued above that pipelining results in additional cache misses every time a packet is passed to a core that uses a different L3 cache (once, in our setup); hence, pipelining outperforms cloning, only if the latter introduces more additional misses due to cache contention.

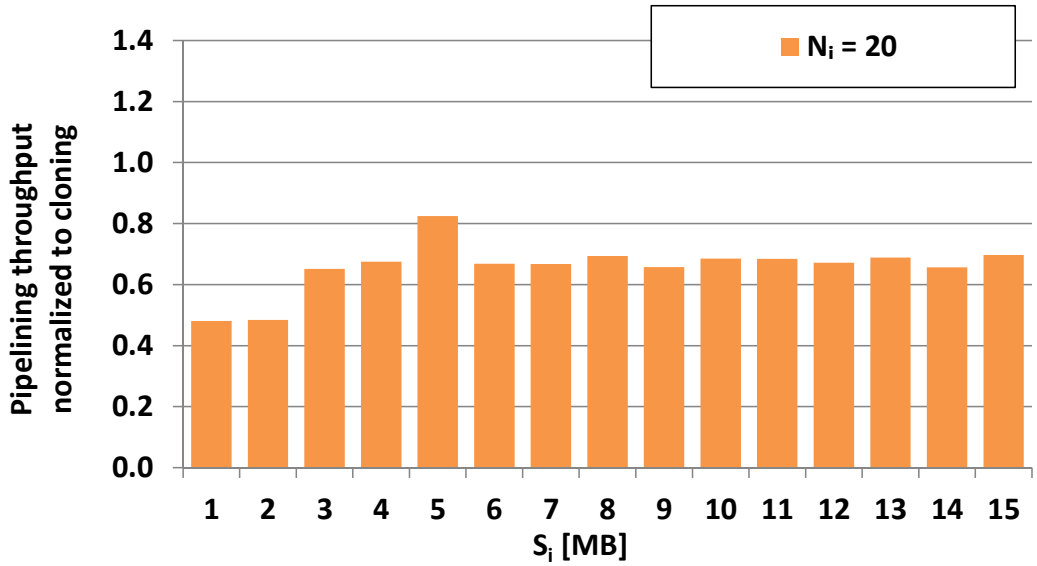

Figure 3.5: Cache contention overhead for $N_i = 20$

We conclude that pipelining does reduce cache contention relatively to cloning, however, this benefit is easily outweighed by its synchronization overhead. Hence, we expect cloning to perform better than pipelining in most realistic scenarios. Pipelining could perform better than cloning only in scenarios that involve large, contention-sensitive data structures and frequent memory accesses.

3.3.4 Pipeline Imbalance Overhead

To work well, pipelining must have balanced pipeline stages, otherwise CPU cycles are wasted since the packet rate through the pipeline is gated by the slowest stage.

To demonstrate this effect, we configured the two elements with parameter values $S_1 = S_2 = 6$MB and $N_1 = 396$, $N_2 = 4$ memory accesses per packet.

Given this setup, the system achieves throughput 728Mbps with cloning and 375Mbps with pipelining, i.e., cloning increases throughput by a factor of 2. This is because, with pipelining, the 4 cores that run the first element perform 99% of the work, while the 4

cores that run the second element perform only 1% of the work.

We conclude that, even when we have large, contention-sensitive data structures, we expect cloning to outperform pipelining if the pipeline stages are imbalanced.

3.3.5 Optimization Problem

We have identified three overheads that can lead to performance differences when parallelizing the same data-flow using either a cloning or a pipelining approach. We now turn to the question of how we can combine the competing factors identified in the previous sections to formulate an optimization problem for choosing the "right" parallelization approach.

Consider a data-flow graph that consists of a sequence of n packet-processing elements and each packet must be sequentially processed by all these elements. We assume that elements do not share any data structures other than socket buffer descriptors and packet contents—packet headers and payloads. Each element i is characterized by:

- $C(i)$ - the number of cycles per packet consumed when running in isolation;
- $S(i)$ - the size of its data structures;
- $N(i)$ - the number of memory requests issued per packet.

We want to run this data flow on a server with m processors, each with multiple cores and a separate last level cache. Each processor j is characterized by:

- B - the budget of CPU cycles;
- LLC - the size of the last level cache.

We want to assign elements to processors, not individual cores: if one core of processor j runs an instance of element i, then all cores of processor j run an instance of element i. Hence, when we say that "processor j runs element i," we mean that each core of processor j runs an instance of element i, and all cores share the same data structure accessed through the processor's LLC. Moreover, we impose the following restriction: any packet received by processor j is processed by all the elements on processor j.

We define the "element placement" matrix A, of dimensions $n \times m$, as follows:

$$A_{ij} = \begin{cases} 1, & \text{if processor } j \text{ runs element } i \\ 0, & \text{otherwise.} \end{cases}$$

We denote by C_j the aggregate number of cycles per packet consumed by processor j, and by $S_j = \sum_i A_{ij} \cdot S(i)$ the aggregate data-structure size of the elements run by processor j.

Incoming traffic is load-balanced across all the processors that run element 1. When processor j receives a packet to be processed according to element i, it does so, then:

- if $i = n$, it sends the packet out;
- else, if processor j runs element $i + 1$, it processes the packet according to element $i + 1$;
- else, it passes the packet to a processor that runs element $i + 1$ (if there are many such processors, it load-balances the traffic across them).

Load-balancing across processors happens based on processor throughput, i.e., processor j gets a fraction of the traffic that is inversely proportional to C_j.

We want to identify the element-placement matrix A that maximizes the platform's throughput R, expressed in packets per second:

$$R = \min_i \left\{ \sum_j \left\{ \frac{B}{C_j} \cdot A_{ij} \right\} \right\} \tag{3.1}$$

This is essentially the throughput of the "slowest" element, i.e., the one that processes the fewest packets per second.

We make three simplifying assumptions:

1. We assume that every LLC miss has a cost of O_m cycles.
2. We assume that inter-processor synchronization (i.e., passing a packet from one processor to another) has a cost of O_s cycles.
3. We assume that each of the S_j memory locations served by the processor's LLC is equally likely to be in the cache. When element i run by processor j accesses its data structure, the probability of a LLC miss is $\pi_j = 1 - \frac{LLC}{S_j}$.

Based on these assumptions, the number of per-packet cycles consumed by the elements run by processor j can be expressed as:

$$C_j = \sum_i C(i) \cdot A_{ij} + O_j^s + O_j^c$$

This is the sum of three components:

- The first component ($\sum_i C(i) \cdot A_{ij}$) represents the aggregate number of cycles needed to process a packet by all the elements that are running on processor j, if they were running in isolation.
- The second component (O_j^s) represents the synchronization overhead. We can approximate this overhead as:

$$O_j^s = \sum_{i=2}^{n} A_{ij} \cdot (1 - A_{i-1j}) \cdot O_s,$$

  where $A_{ij} \cdot (1 - A_{i-1j})$ is the number of times processor j gets each packet from another processor and O_s is the cost spent for inter-processor synchronization, expressed in number of cycles.
- The third component (O_j^c) represents the cache-contention overhead. We can approximate this overhead as:

$$O_j^c = \sum_i A_{ij} \cdot \pi_j \cdot N(i) \cdot O_m,$$

  where $\pi_j \cdot N(i) \cdot O_m$ is the expected number of cycles spent dealing with cache misses due to contention.

To sum up, Equation 3.1 specifies the throughput of the system in scenarios where the bottleneck is the CPU (i.e., the server starts dropping packets because it does not have enough processing cycles to perform the necessary per-packet computation and/or memory accesses). For this we assume that CPU cycles are spent either performing per-packet computation or dealing with cache misses resulting from inter-processor synchronization and cache contention.

3.4 Summary

In this chapter we analyze how a software packet-processing platform can exploit the increasing level of parallelism present in the modern multicore servers. More concretely, we show how packet processing should be parallelized among multiple cores in order to maximize the system's performance.

One possibility is the *"cloning"* approach, where each packet is handled by a single core that reads the packet from memory and performs all the necessary processing steps. Another possibility is the *"pipelining"* approach, where each packet is handled by multiple cores: one core reads it from memory, then passes it to another core for the first processing step, which passes it to another core for the second processing step, and so on.

The most recent general-purpose networking projects use the cloning approach, because it yields higher performance [34, 43]. However, a common criticism is that this is the case only for the simple, uniform workloads considered by these projects.

At a first glance, choosing between the two approaches involves a trade-off: on the one hand, the cloning approach avoids passing the packet between different cores, hence eliminating synchronization and introducing fewer compulsory cache misses per packet; on the other hand, it requires that each core performs all the processing steps for each packet, which may introduce a higher number of avoidable cache misses per packet due to cache contention—each core accesses multiple data structures that might not fit in the cache. Hence, it seems intuitive that each approach would be best suited for different packet-processing applications—e.g., the cloning approach would be better for simple applications, while the pipelining approach would be better for applications with many processing steps and large cacheable data structures.

After extensive experiments, we concluded that, in practice, there is no real trade-off between the two approaches: the cloning approach is always better. This is because pipelining introduces several kinds of overhead that end up outweighing its potential benefit. For instance, memory recycling, passing socket-buffer descriptors and packet contents between different cores results in 13–14 extra compulsory cache misses per packet.

It *is* possible to craft a synthetic workload that performs better under the pipelining approach: it has to be a workload with enough processing steps and the *right* size of cacheable data structures such that running it using the cloning approach results in more than 15 extra avoidable cache misses per packet than running it using the pipelining approach. We describe such a workload in Section 3.3.3: each received packet triggers more than 200 random memory accesses to a data structure that is exactly double the size of an L3 cache. However, even a *small* deviation from these numbers causes the advantage of the pipeline over the parallel approach to disappear.

4 Predictable Performance

Prediction is very difficult, especially if it's about the future.

Niels Bohr

4.1 Introduction

In recent years, both practitioners and researchers have argued for building evolvable networks, whose functionality changes with the needs of its users and is not tied to particular hardware vendors [9, 15, 43, 56]. An inexpensive way of building such networks is to run a network-programming framework like Click [52] on top of commodity general-purpose hardware [15, 43, 56]. Sekar et al. recently showed that, in such a network, operators can reduce network provisioning costs by up to a factor of 2.5 by dynamically consolidating middlebox functionality, i.e., assigning packet-processing tasks to the available general-purpose devices in order to minimize resource consumption [66].

To become a credible alternative to specialized hardware, general-purpose networking needs to offer not only flexibility but also predictable performance: network operators are unlikely to accept the risk that an unlucky configuration could cause unpredictable drop in network performance, potentially leading to customer dissatisfaction and violations of service-level agreements.

Several projects have demonstrated that general-purpose multicore hardware can perform packet processing at line rates of 10Gbps or more [34, 43, 44, 56, 57]. However, in all cases, this was achieved under a crucial simplifying assumption of uniformity: all processing cores see the same type/amount of traffic and run identical code, while all packets receive the same kind of conventional packet processing (e.g., IP forwarding or some particular form of encryption). This setup allowed for low-level tuning of the entire system to one

particular, simple, uniform workload (e.g., manually setting buffer and batch sizes).

Building a general-purpose system that offers predictable performance is considered a hard problem, especially when this system needs to support an evolvable set of applications that are potentially developed by various third parties. Such a system may perform as expected under certain conditions, but then a change in workload or a software upgrade could cause unpredictable, potentially significant, performance degradation.

One important reason for the lack of predictability is the complicated ways in which software processes running on the same hardware affect each other:

- false sharing [21];
- unnecessarily shared data structures [23];
- contention for shared hardware resources [70].

In particular, contention for shared hardware resources (caches, memory controllers, buses) has been the subject of extensive research for more than two decades: researchers have been working on predicting the effects of resource contention since the appearance of simultaneous multithreaded processors [13, 27, 30, 68, 69, 73, 75]. Yet, to the best of our knowledge, none of the proposed models have found their way into practice. And packet-processing workloads create ample opportunity for resource contention, as they move packets between network card, memory controller and last-level cache, all of which are shared among multiple cores in modern platforms.

In this chapter, we design and build a packet-processing system that combines both *ease of programmability* and *predictable performance*, while supporting a diverse set of applications and serving multiple clients, each of which may require different types/combinations of packet processing. To offer ease of programmability, we rely on the Click network-programming framework [52]. We do not introduce any additional programming constraints, operating-system modifications, or low-level tuning for particular workloads.

In particular, we present a Click-based packet-processing system, built on top of a 12-core Intel Westmere platform, that supports a diverse set of realistic packet-processing applications (Section 4.2). Given this setup, we first investigate how resource contention affects performance in the packet-processing context (Section 4.3) and whether we can predict the effects of resource contention using simple techniques (Section 4.4). We also explore whether it makes sense to use a "smart" strategy for application-to-core placement (i.e., avoid scheduling together the applications that are likely to contend for hardware resources), in order reduce the effects of resource contention and maximize the performance of the system (Section 4.5).

More concretely, in this chapter, we answer the following questions:

- Does resource contention cause significant performance drop?
- How does the performance drop depend on specific properties of the applications?
- Can we predict the effects of resource contention in a practical manner?
- What do we stand to gain by using "contention-aware scheduling" [77]?

We show that it is feasible to build a software packet-processing system that achieves predictable performance in the face of resource contention. More specifically, we show that by using simple offline profiling of each application running alone, we are able to predict the contention-induced performance drop suffered by each of the applications sharing the system, with an error smaller than 3%.

We also show that contention-aware scheduling may not be worth the effort in the context of packet processing. More specifically, in our system, the maximum overall performance improvement achieved by using contention-aware scheduling is 2%—and that only in one particular corner case.

We provide intuition behind these results, and we quantitatively argue that they are not artifacts of the Intel architecture, rather they should hold on any modern multicore platform.

4.2 System Setup

In this section, we first describe the hardware configuration (Section 4.2.1) and the software configuration of our platform (Section 4.2.2); then we introduce the packet-processing applications that we use to evaluate our work (Section 4.2.3).

4.2.1 Hardware Configuration

As a basis for our system, we use a 12-core Intel Westmere general-purpose server, illustrated in Figure 4.1. The server is equipped with two Intel Xeon 5660 processors, each with 6 cores running at 2.8GHz and an integrated memory controller. The 6 cores of each processor share a 12MB L3 cache, while each core has a private L2 cache (256KB) and a private L1 cache (32KB for instructions and 32KB for data). The two processors are interconnected via a 6.4GT/sec QuickPath interconnect (QPI). The server has 6 DDR3 memory modules (2GB each, 1333MHz) and 3 dual-port 10Gbps network interface cards (NICs) that use the Intel 82599 Niantic [3] chipset, resulting in 6×10Gbps ports. The server runs SMP-Click [29] version 1.7, on Linux kernel 2.6.24.7.

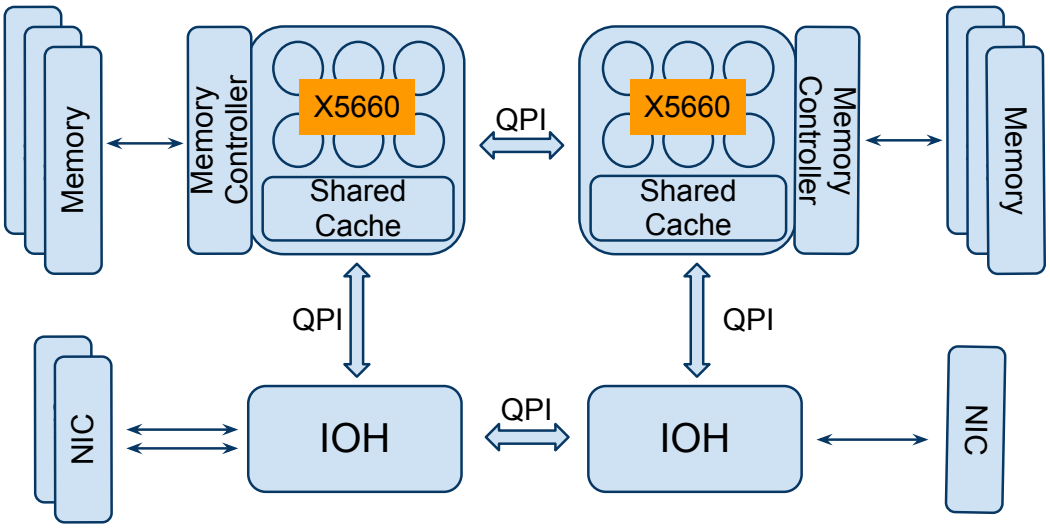

Figure 4.1: Overview of our platform's hardware architecture.

4.2.2 Software Configuration

Packet-Processing Parallelization

An important question when configuring a software platform is how packet processing should be parallelized (i.e., use a cloning approach, a pipelining approach, or a mix of the two). We discuss this problem in detail in Chapter 3 and we conclude that, for realistic packet-processing workloads, the cloning approach always results in better performance. And indeed, even though it is possible to create a synthetic workload that performs better under the pipelining approach (see Section 3.3.3), none of the realistic workloads that we looked at (see Section 4.2.3) comes even close to such behavior.

We adopt the *cloning* approach for our system. Traffic arriving at each of our N network ports is split into Q receive queues. We refer to all traffic arriving at one receive queue as a *flow*. This is traffic that corresponds to one set of clients of our networking platform, all of which require the same type of packet processing. Each flow is handled by one core, which is responsible for reading the flow's packets from their receive queue, performing all the necessary processing, and writing them to the right transmit queue. Each core reads from its own receive queue(s) and writes to its own transmit queue(s), which are not shared with other cores. So, we have $N \cdot Q$ flows, each one assigned to one core, and each flow potentially involving a different type of packet processing.

In this chapter, we focus on the scenario where each core processes one packet-processing flow: we use $N = 6$ ports and $Q = 2$ receive queues per port, so we have 12 different flows, each assigned to a separate core (we discuss this choice in Section 4.6). However, the

Niantic cards support up to $Q = 128$ receive queues, so our prototype can, in principle, support hundreds of different flows.

NUMA Memory Allocation

We ensure that each flow accesses its data "locally," i.e., through the memory controller that is directly connected to the processor handling the flow. We do this for two reasons: first, it has been shown and we have also verified experimentally that accessing data remotely has a significant impact on memory-access latency [19], which, in our context, leads to significant performance degradation. Second, to access data remotely, a flow has to use the processor interconnect, which can become a significant contention factor [19,77].

Theoretically, there are two scenarios where we might not be able to ensure local memory access, but these do not arise in our system and we explain next why this is the case:

- Consider two flows, f_1 and f_2, that run on different processors and access the same data structure; one of the two flows will have to access the data remotely. This scenario does not arise in our system: if there are multiple flows that need to access the same data structure, we run all of them on the same processor. If there are more flows than per-processor cores that need to access the same data structure, we replicate the data structure across memory domains. We acknowledge that, in principle, such replication may break the semantics of a packet-processing application, however, we have not yet encountered any such realistic case. The closest we came was the redundancy-elimination application (described in Section 4.2.3), but, even there, it turned out that all the relevant data structures could be replicated across memory domains.

- Consider a flow f running on a core of processor P_1, accessing its data locally; due to contention-aware scheduling [19, 77], we decide to move this flow to a core of processor P_2; as a result, f must now access its data remotely. This scenario does not arise in our system either: we argue that, in our context, it does not make sense to perform contention-aware scheduling and one of the resulting benefits is that we do not have to deal with remote memory accesses.

Avoidable Contention in the Software Stack

Before setting out to study resource contention between applications running on different cores, we sought to eliminate any form of "underlying" resource contention from our system, i.e., contention introduced not by the applications themselves but by the design of the underlying software stack: NIC driver, operating system, Click.

We identified two sources of such contention in our system:

- false sharing;
- unnecessary data sharing among multiple cores (e.g., the book-keeping data structures in the Niantic driver and the random seed of the Click random number generator were shared among multiple cores).

We eliminated the former by padding data structures appropriately and the latter by replicating per-core data structures. Similar problems and fixes were recently presented in a scalability analysis of the Linux kernel [23].

4.2.3 Packet-Processing Workloads

We designed our workloads to involve realistic forms of packet processing but make the job of predicting their performance as hard as possible. We implemented 5 forms of packet processing that are deployed in current network devices and cover a wide range of memory and CPU behavior. In our experiments, we craft the traffic that is processed by the system so as to maximize resource contention.

Specifically, we implemented the following types of packet processing:

- *IP forwarding (IP).* Each packet is subjected to full IP forwarding, including longest-prefix-match lookup, checksum computation, and time-to-live (TTL) update. We use the RadixTrie lookup algorithm provided with the Click distribution and a routing-table of 128 000 entries. As input, we generate packets with random destination addresses, because this maximizes IP's sensitivity to contention.
- *Monitoring (MON).* In addition to full IP forwarding, each packet is further subjected to NetFlow [2], a monitoring application. NetFlow collects statistics as follows: it applies a hash function to the IP and transport-layer header of each packet, uses the outcome to index a hash table with per-TCP/UDP-flow entries, and updates a few fields (a packet count and a timestamp) of the corresponding entry. As input, we generate packets with random IP addresses, such that the NetFlow hash table contains 100 000 entries. MON is a representative form of memory-intensive packet processing that benefits significantly from the L3 cache since both the routing table and the NetFlow hash table are cacheable data structures. Also, it captures the nature of a wide range of packet-processing applications: applying a hash function to a portion of each packet and using the outcome to index and update a data structure.
- *Small firewall (FW).* In addition to full IP forwarding and NetFlow, each packet is further subjected to filtering: each packet is sequentially checked against 1000 rules and, if it matches any, it is discarded. We use sequential search (as opposed to a more sophisticated algorithm) because we consider a relatively small number

of rules that can fit in the L2 cache. As input, we generate packets with random IP addresses that never match any of the rules; as a result, each packet is checked against all the rules, which maximizes FW's sensitivity to contention. This is a representative form of packet processing that benefits significantly from all the levels of the cache hierarchy.

- *Redundancy elimination (RE).* In addition to full IP forwarding and NetFlow, each packet is further subjected to RE [67], an application that eliminates redundant traffic. RE maintains a "packet store" (a cache of recently observed content) and a "fingerprint table" (that maps content fingerprints to packet-store entries). When a new packet is received, RE first updates the packet store, then uses the fingerprint table to check whether the packet includes a significant fraction of content cached in the packet store; if yes, instead of transmitting the packet as is, RE transmits an encoded version that eliminates this (recently observed) content. The assumption is that the device located at the other end of the link maintains a similar packet store and is able to recover the original contents of the packet. We implemented a packet store that can hold 1 second's worth of traffic and a fingerprint table with more than 4 million entries. This is a representative form of memory-intensive packet processing that does not significantly benefit from caching.

- *Virtual private network (VPN).* Each packet is subjected to full IP forwarding, NetFlow and AES-128 encryption. This is a representative form of CPU-intensive packet processing.

- *Synthetic processing (SYN).* For each received packet, we perform a configurable number of CPU operations (counter increments) and read a configurable number of random memory locations from a data structure that has the size of the L3 cache. We use this for profiling. We denote by *SYN_MAX* the most aggressive synthetic application that we were able to run on our system, which performs no other processing but consecutive memory accesses at the highest possible rate.

In Table 4.1 we summarize the characteristics of each of these types of packet processing during a "solo" run—one core runs the packet-processing type, while all the other cores are idle. In order to collect profile information, we rely on the hardware performance counters present on the Intel platform. We use Oprofile [5] to count instructions, L2 hits, L3 references, L3 misses, while the L3 hits is computed as the difference between the references and misses.

4.3 Understanding Resource Contention

In this section, we quantify the effects of resource contention for packet-processing flows (Section 4.3.1), we identify which resources flows mostly contend for (Section 4.3.2) and

Flow type	cycles per instruction	L3 references per second (millions)	L3 hits per second (millions)	cycles per packet	L3 references per packet	L3 misses per packet	L2 hits per packet
IP	1.33	25.85	20.21	1813	14.64	3.19	18.58
MON	1.43	27.26	21.32	2278	19.40	4.23	19.58
FW	1.63	2.71	2.13	23 907	20.22	4.29	56.10
RE	1.18	18.18	5.52	27 433	155.87	108.51	45.63
VPN	0.56	9.45	7.08	8679	25.63	6.41	30.71

Table 4.1: Characteristics of each type of packet processing during a solo run.

which flow properties determine the level of contention (Section 4.3.3), and we provide intuition behind our observations (Section 4.3.4).

We observe that the contention-induced performance drop suffered by a packet-processing flow is mostly determined by the number of last-level cache references per second performed by other flows sharing the same cache—not so much by the particular types of packet processing performed by these flows. We provide intuition behind this behavior and quantitative evidence that it is not particular to our platform—it is simply the result of flows sharing a last-level cache, and it should hold on any modern multicore platform. As we will see, this observation is what enables us to do simple yet accurate performance prediction (Section 4.4).

In terms of terminology and notation, when we use the term "cache," we refer to the last-level cache shared by all cores of the same processor unless otherwise specified. We use "cache refs/sec" as an abbreviation for "cache references per second." We say that a flow *co-runs* with other flows when they all run on different cores of the same processor; we refer to all these flows as *co-runners*. In each experiment, we typically co-run 6 flows and study the performance drop suffered by one of these flows due to contention; we denote this flow by T (for "target") and each of its co-runners by C (for "competitor"). With respect to a flow T, we use the term *competing references* to refer to all the last-level cache references performed by this flow's co-runners.

In all our experiments, we compute the performance drop suffered by a flow T due to contention with a set of competing flows as follows: first, we measure the throughput τ_s achieved by flow T during a solo run. Then we measure the throughput τ_c achieved by flow T when it co-runs with the set of competing flows. The contention-induced performance drop suffered by flow T is $\frac{\tau_s - \tau_c}{\tau_s}$. Each data point in our graphs represents the average over 5 independent runs of the same experiment (the variance is negligible).

4.3.1 Effects of Resource Contention

We start by measuring the contention-induced performance drop suffered by realistic flow types in different scenarios.

For each pair of realistic flow types X and Y, we run an experiment in which a flow of type X co-runs with 5 flows of type Y. In Figure 4.2 we show the performance drop suffered due to contention by the flow of type X. For example, when a VPN flow co-runs with 5 IP competitors, it suffers a drop of 12.25%.

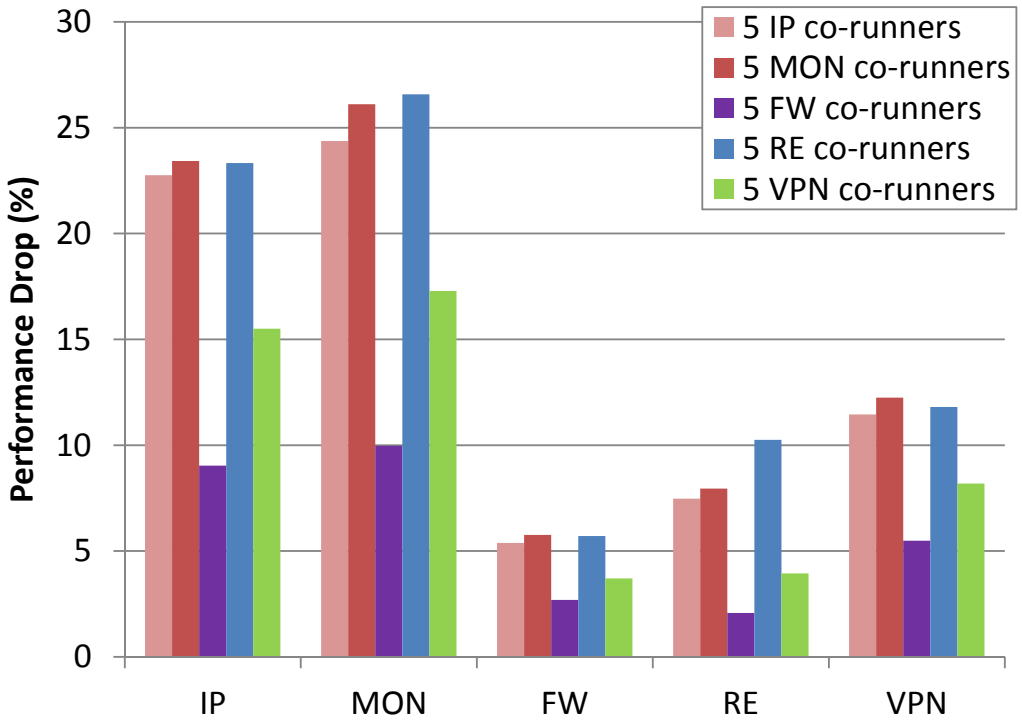

Figure 4.2: The effects of resource contention: performance drop suffered by each realistic flow of type X when co-running with 5 realistic flows of type Y.

In Figure 4.3 we also show the average performance drop suffered by each flow type across all 5 scenarios that involve a target flow of that type. For example, the average performance drop suffered by the MON flow across all 5 scenarios that involve a target MON flow is 20.86%.

We observe that MON is the most sensitive flow type, suffering a performance drop of up to 27% (highest bar in Figure 4.2), while FW suffers less than 6% in all experiments. RE is the most aggressive flow type, causing a performance drop of up to 27%, while FW causes a performance drop of less than 10% in all experiments.

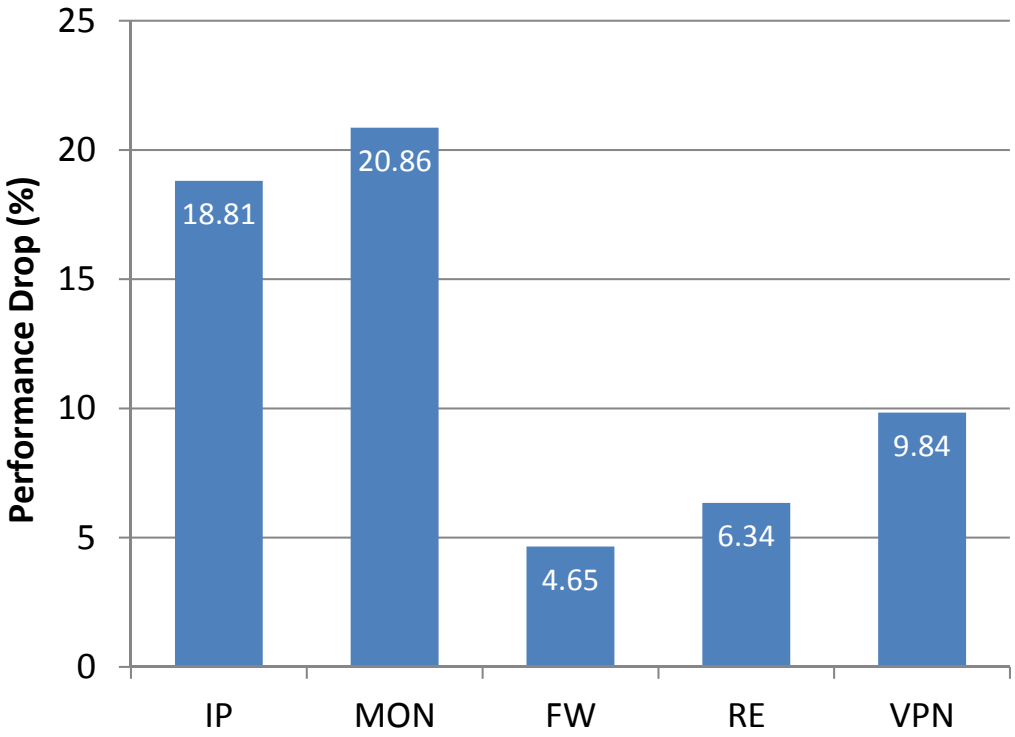

Figure 4.3: The effect of resource contention: average performance drop suffered by each realistic flow type across all 5 scenarios that involve a target flow of that type.

To draw meaningful conclusions from these numbers, we need to understand what are the properties of a packet-processing flow that make it sensitive and/or aggressive with respect to contention.

4.3.2 Contended Resources

We first identify which resources are responsible for contention. There are two candidates: the cache and the memory controller. We can rule out the processor interconnect, because our configuration ensures that each flow accesses its data locally, hence does not use the interconnect (Section 4.2.2).

To assess the level of contention for the two candidate resources, we use the three system configurations illustrated in Figure 4.4, Figure 4.5 and Figure 4.6, where T denotes the target flow (whose performance drop we are measuring), $M(T)$ denotes flow T's data structures, C denotes a competing flow, and $M(C)$ denotes the corresponding data structures of the competing flow. We highlight the resource that is contended in each of these configurations.

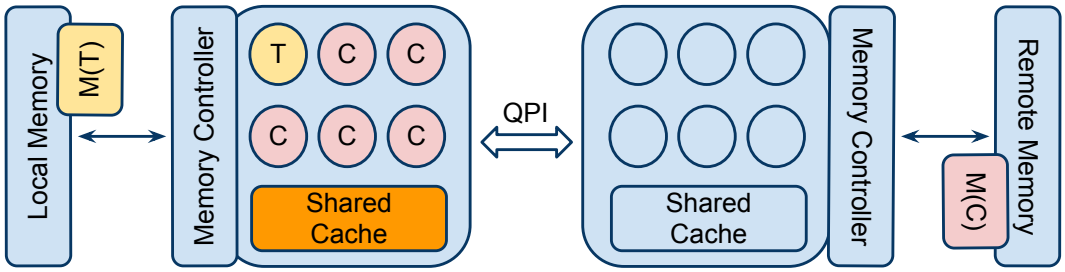

Figure 4.4: Configuration that exposes contention only for the cache.

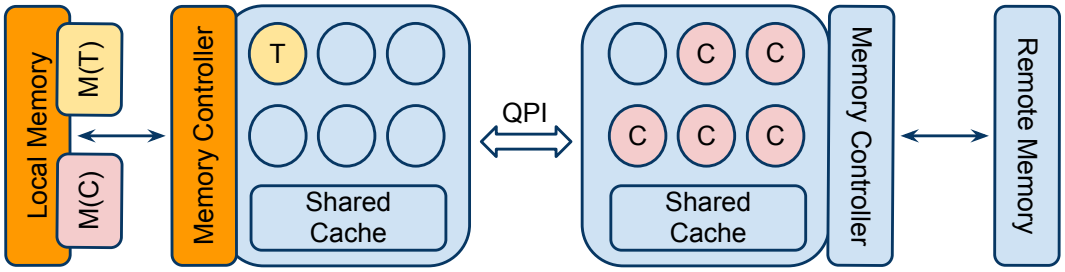

Figure 4.5: Configuration that exposes contention only for the memory controller.

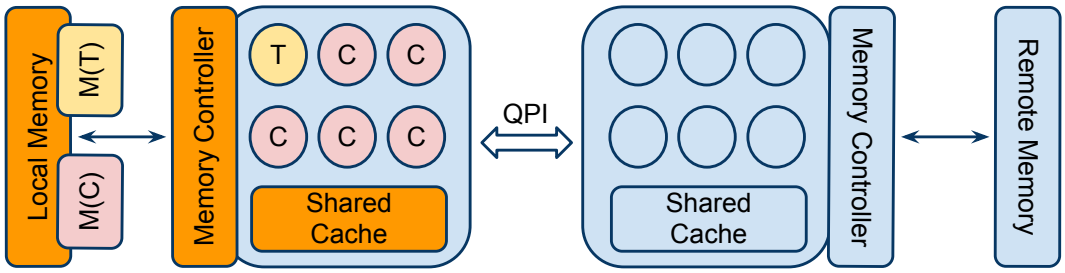

Figure 4.6: Configuration that exposes contention for both the cache and the memory controller.

Each configuration allocates processing cores and memory to the co-running flows, so as to expose contention for different resources [19]:

- The first configuration (Figure 4.4) creates contention only for the cache: T contends with Cs only for the L3 cache; Cs' data is remote, hence accessed through a different memory controller.
- The second configuration (Figure 4.5) creates contention only for the memory controller: T contends with Cs only for the memory controller; Cs run on a different processor, hence use a different L3 cache.

- The third configuration (Figure 4.6) creates contention for both the cache and the memory controller: T contends with the Cs for both the L3 cache and the memory controller.

In each configuration, we measure the contention-induced performance drop suffered by realistic flow types when they encounter different levels of competition. For each realistic flow type X, we co-run a flow of type X with 5 flows of type SYN multiple times, ramping up the number of cache refs/sec performed by the SYN flows. We measure the performance drop suffered by the flow of type X as a function of the competing cache refs/sec performed by the SYN flows.

We show the effects of contention for different resources in Figure 4.7 (contention for the cache), Figure 4.8 (contention for the memory controller) and Figure 4.9 (contention for both the cache and the memory controller).

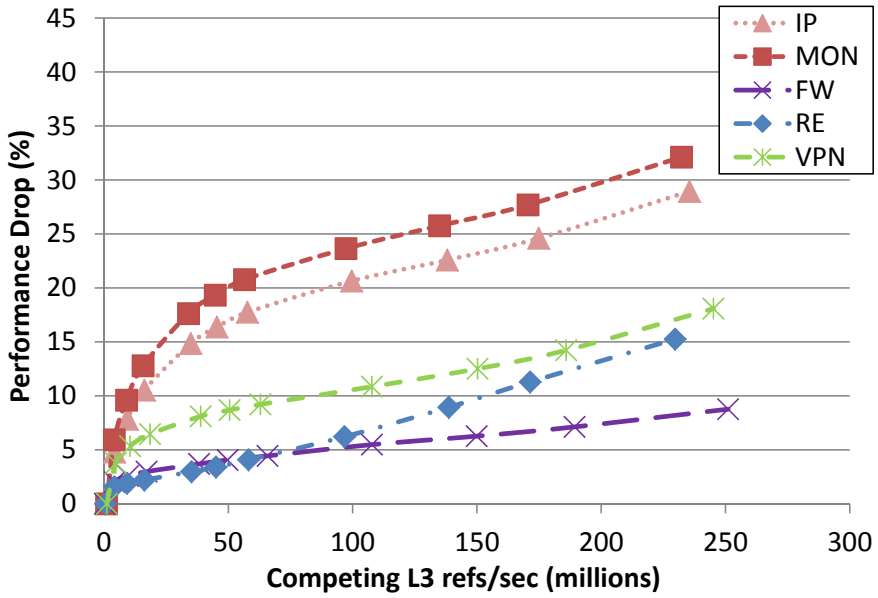

Figure 4.7: Contention for the cache. Performance drop suffered by each realistic flow type in the configuration of Figure 4.4.

The results show that the dominant contention factor is the cache. The most sensitive flow type (MON) suffers up to 32% when competing only for the cache (the curve with square data points in Figure 4.7), up to 6% when competing only for the memory controller (the curve with square data points in Figure 4.8), and up to 40% when competing both for the cache and the memory controller (the curve with square data points in Figure 4.9). When competing for both resources, a flow suffers more than the sum of its drop in the other two configurations due to a compounding effect: cache contention causes the flows

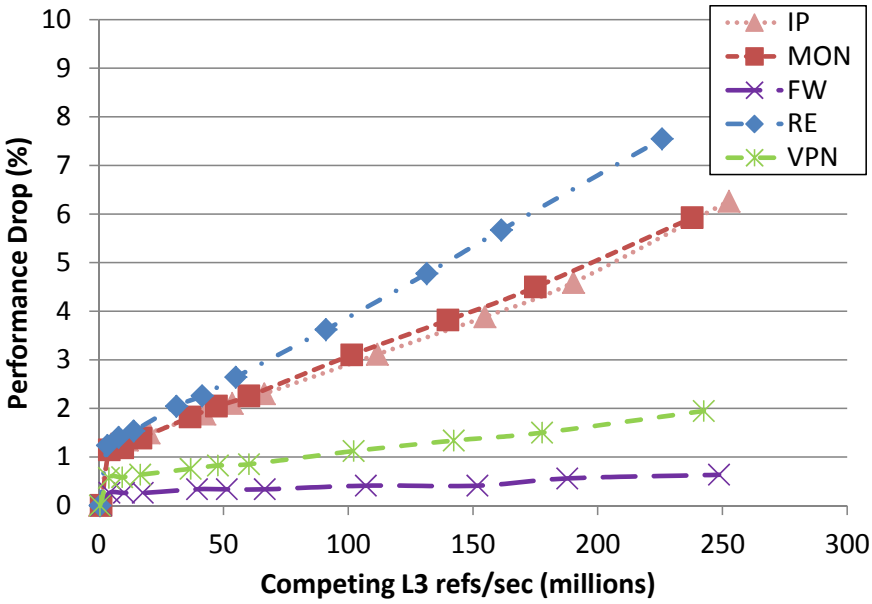

Figure 4.8: Contention for the memory controller. Performance drop suffered by each realistic flow type in the configuration of Figure 4.5.

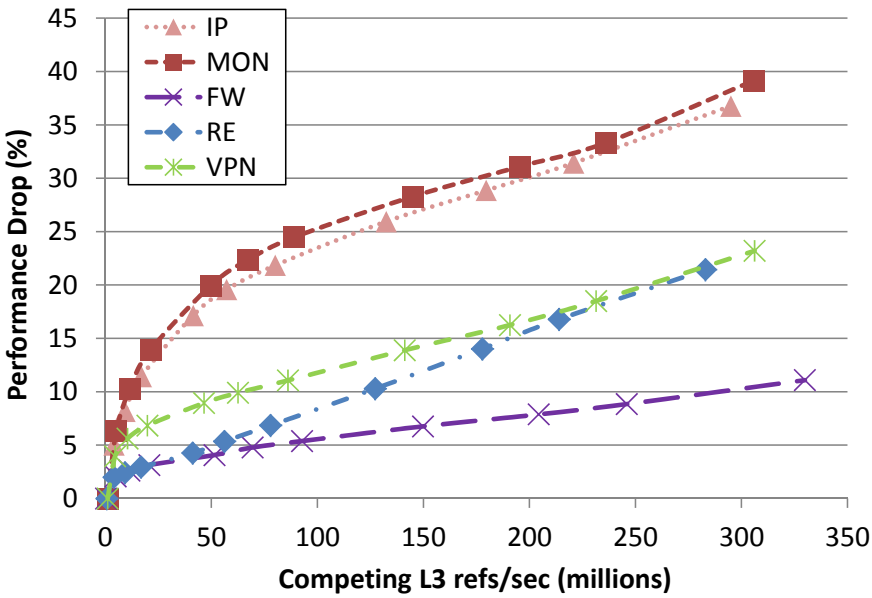

Figure 4.9: Contention for both the cache and the memory controller. Performance drop suffered by each realistic flow type in the configuration of Figure 4.6.

to incur more cache misses, hence creates more memory accesses, which, in turn, create more contention for the memory controller.

Our conclusion relates to prior work as follows: it differs from the conclusion drawn by running SPEC benchmarks on multicore platforms—in that case, the dominant contention factors were found to be the memory controller and the processor interconnect [19, 77]. The difference may come from the fact that packet-processing workloads benefit from the cache more than SPEC benchmarks and/or the fact that, in our context, overloading the interconnect is unnecessary. Our conclusion is consistent with recent results on software routers: in a software router running on an Intel Nehalem platform, as long as we have sufficient network I/O capacity, the bottleneck lies with the CPU and/or memory latency [34].

4.3.3 Sensitivity and Aggressiveness

We now look at which properties of a packet-processing flow determine its sensitivity and aggressiveness, i.e., the amount of damage that it suffers from its co-runners and the amount of damage that it causes to them.

First, we observe a positive correlation between the sensitivity of a flow and the number of cache hits per second that it achieves during a solo run. Figure 4.3 shows that the higher the number of hits per second achieved by a flow type during a solo run (Table 4.1), the higher the average performance drop suffered by the flow. This makes sense: sharing a cache with co-runners causes memory references that would result in cache hits (if the flow ran alone) to become cache misses; the more hits per second a flow achieves during a solo run, the more opportunity there exists for these hits to become misses, leading to higher performance drop.

Second, we observe that the amount of damage suffered by a given flow is mostly determined by the number of competing cache refs/sec, not so much by the types of competitors. Differently said, two flows, C_1 and C_2, that perform the same number of cache refs/sec will cause *roughly* the same performance drop to a given co-runner, regardless of the type of packet processing performed by C_1 and C_2.

This can be seen in Figure 4.10 (a merge of the results in Figure 4.2 and Figure 4.9), which shows the performance drop suffered by each realistic flow type when it co-runs with SYN flow competitors (curves) as well as realistic flow competitors (individual points). For example, the curve MON(S) shows the performance drop suffered by a MON flow when it co-runs with SYN flows, while the individual squares MON(R) show the performance drop suffered by a MON flow when it co-runs with various realistic flow types. Each MON(R) square corresponds to a different realistic competitor type.

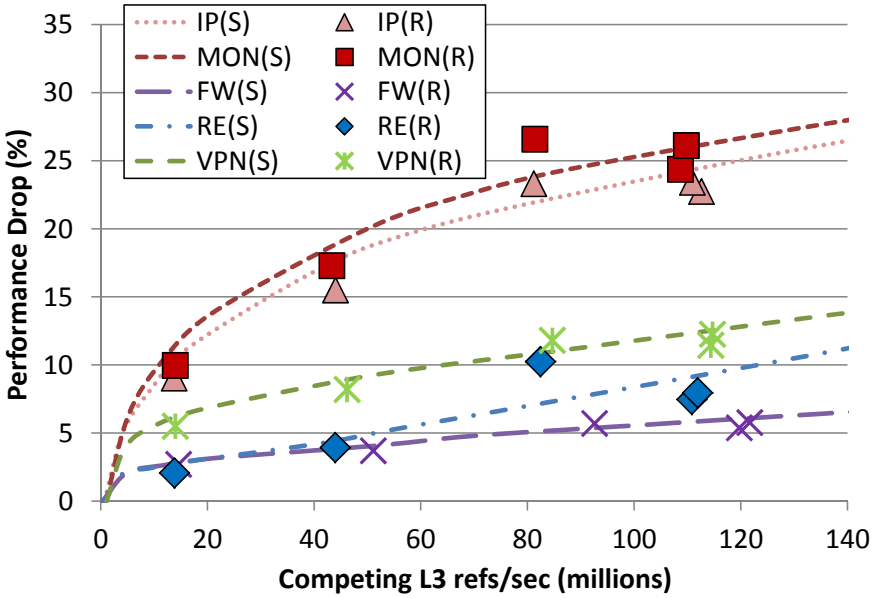

Figure 4.10: Performance drop suffered by each realistic flow as a function of the competing cache refs/sec when it co-runs with SYN flows as well as realistic flows.

We can observe that a MON flow suffers a 27% drop when competing with 5 RE flows that generate 80 million cache refs/sec, and it suffers a 24% drop when competing with 5 SYN flows that generate the same number of cache refs/sec. So, RE flows cause about the same damage with SYN flows that generate the same rate of cache references, even though RE involves redundancy elimination, whereas SYN involves random memory accesses.

We found this observation partly intuitive and partly surprising. The intuitive part is that more competing cache references result in more damage, because they reduce the effective cache space of the target flow. The surprising part is that the particular memory access pattern of the competitors is not significant, and instead the rate of competing cache references mostly determines the amount of damage suffered by flows. It is worth noting that most of the complexity of existing mathematical models that predict contention effects comes from their effort to characterize the memory access patterns of the co-runners and their interaction.

Third, we observe that the performance drop of a sensitive flow drops sharply at first as we increase the number of competing cache refs/sec, however, beyond some point, the drop slows down significantly. For instance, as we can see in Figure 4.10, the performance of a MON flow drops by 20% when competition goes from 0 to 50 million cache refs/sec, but only an extra 5% when competition goes from 50 to 100 million cache refs/sec. As a result, the performance of a MON flow drops roughly the same, whether it is co-running

with IP, MON, or RE competitors, since all these flows contribute at least 50 million cache refs/sec.

In summary, we made three observations:

- A flow's sensitivity to resource contention depends on the number of cache hits/sec that the flow achieves during a solo run.
- The specific amount of damage that a flow suffers due to contention is mostly determined by the number of cache refs/sec performed by its competitors, and not by the exact type of packet processing that they perform.
- The performance of a sensitive flow at first drops sharply as the number of competing cache refs/sec increases; however, once a "turning point" is reached, the performance drop suffered by each sensitive flow stays within a relatively small range.

4.3.4 Explanation of Our Observations

Before we show how we can leverage these observations in order to predict the effects of resource contention, we provide intuition for them. We concentrate on cache contention, since this is the dominant contention factor in our system and we show that our observations can be explained as the result of multiple processes sharing a last-level cache and they are not the result of a particular artifact of our platform, but a general property of modern server architectures.

Sensitivity Depends on Cache Hits/Sec

We can express the performance drop suffered by a flow due to cache contention as follows:

- Suppose the flow achieves h cache hits/sec and processes n packets/sec during a solo run.
- Suppose that, due to contention, the flow suffers *hit-to-miss conversion rate* κ, i.e., each memory reference that was a hit during a solo run turns into a miss with probability κ.
- Without contention, processing n packets takes 1 second. With contention, processing n packets results in $\kappa \cdot h$ extra cache misses and takes $1 + \delta \cdot \kappa \cdot h$ seconds, where δ is the extra time needed to complete a memory reference that is a cache miss instead of a cache hit.

- Hence, the performance drop suffered by the flow in terms of packets/sec can be expressed as

$$\frac{n - \frac{n}{1+\delta\kappa h}}{n} = \frac{1}{1 + \frac{1}{\delta\kappa h}}. \tag{4.1}$$

Performance drop increases with competition (for the cache), primarily because the hit-to-miss conversion rate increases with competition. The value of δ provided by our platform's specifications is 43.75 nanoseconds—although, in practice, its exact value depends on the nature of memory accesses and also slowly increases with competition.

In a worst case scenario, the hit-to-miss conversion rate is $\kappa = 1$, i.e., all of the cache hits achieved by the target flow during a solo run turn into misses due to contention. Figure 4.11 shows this worst-case performance drop as a function of the number of cache hits/sec achieved by the flow during a solo run, for different values of δ. For instance, if a packet-processing flow achieves fewer than 20 million cache hits/sec during a solo run, even if all the hits turn into misses, the flow's performance cannot drop by more than 47% (assuming $\delta = 43.75$ nanoseconds).

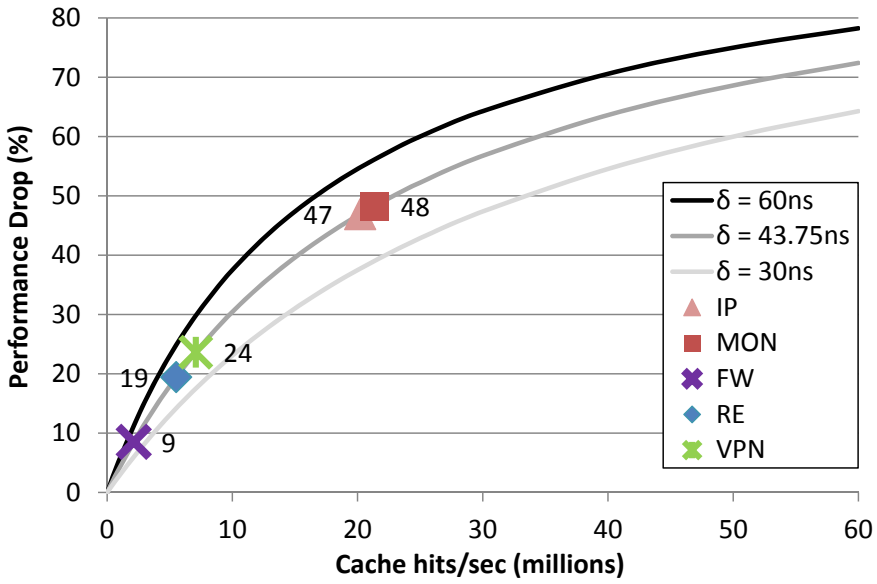

Figure 4.11: Estimated maximum performance drop suffered by a flow as a function of the cache hits/sec it achieves during a solo run. The estimates are based on Equation 4.1, for $\kappa = 1$ and different values of δ. The graph also shows the data points that correspond to our realistic packet-processing flows, assuming $\delta = 43.75$ nanoseconds. For example, the maximum performance drop that could be suffered by an IP flow is 47%.

As a side note, according to Equation 4.1, a flow's worst-case performance drop depends only on the hits/sec achieved by the flow during a solo run, not by other characteristics

of the flow, such as cycles spent on computation or total memory references per second. This is what makes hits/sec a good metric for a flow's worst-case sensitivity to contention.

Aggressiveness Determined by Cache Refs/Sec

We observed that, in our setup, the aggressiveness of a set of flows is mostly determined by their cache refs/sec: a set of realistic flows and a set of SYN flows that perform the same number of cache refs/sec cause roughly the same damage to a given target flow T.

We explain this as follows: each of our realistic packet-processing flows accesses at least a few megabytes of data. When several of these flows co-run, they access a total amount of data that is significantly larger than the cache size, which causes them to access the cache close to uniformly. As a result, from the point of view of a target flow T that shares the cache with these flows, they behave similarly to a set of SYN flows (that access the cache uniformly by construction).

In Section 4.6, we briefly discuss the scenario where the working-set sizes of the competing flows are relatively small such that the cache is not saturated, and explain why we do not address this scenario.

Shape of Performance Drop Curve

To understand how performance drop changes with competition, we look at how the hit-to-miss conversion rate changes with competition.

Figure 4.12 shows the conversion rate suffered by a MON flow as a function of cache competition, as we measure it on our platform (using the configuration of Figure 4.4) and as we analytically estimate it using a simple model. We will use the model to provide intuition (*not* accurate prediction), then discuss how it matches the measured data.

We describe the model in the Appendix (A) and summarize here the gist: we have a target flow T that shares a cache with a set of competitors.

- Consider a sequence of cache references, $\langle t, c_1, c_2, \ldots, c_Z, t' \rangle$, where: t and t' are two consecutive references performed by flow T to the same cache line, t' was a hit during a solo run, and $c_i, i = 1..Z$, are the competing references that occur between t and t'.

- Suppose that each competing reference c_i evicts the content cached by t with probability p_{ev}, independently from any other competing reference. t' is a hit if none of the Z competing references evict this content, i.e., $P(hit|Z) = (1 - p_{ev})^Z$. The target flow's hit-to-miss conversion rate is $1 - P(hit)$.

- Suppose that each reference that occurs after t is: either a competing reference, with probability p_c, or t', with probability $p_t = 1 - p_c$. Hence, Z is a random variable of geometric distribution with success probability p_t, i.e., $P(Z = z) = (1 - p_t)^z p_t$.

To compute $P(hit)$ as a function of competition, we need to know how p_{ev} and p_t change with competition. The following assumptions allow us to approximate them: (a) the competitors access the cache uniformly, (b) the target flow accesses its data uniformly, and (c) the target flow and the competitors have similar sensitivity to contention, i.e., suffer approximately the same hit-to-miss conversion rate.

In Figure 4.12 we show the measured and the estimated hit-to-miss conversion rate suffered by a MON flow that shares the cache with SYN competitors, as a function of the competing cache refs/sec. We also show the measured conversion rate suffered by separate functions of the MON flow ("flow_statistics" performs all the NetFlow-specific processing; "radix_ip_lookup" performs IP-table lookups; "check_ip_header" checks whether each packet has a valid IP header; "skb_recycle" performs memory management).

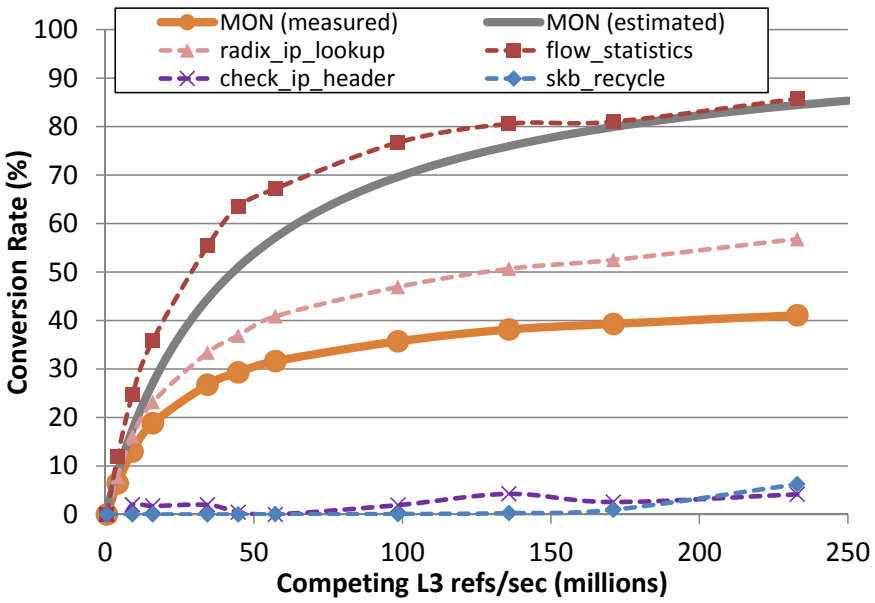

Figure 4.12: Measured and estimated hit-to-miss conversion rate suffered by a MON flow that shares the cache with SYN competitors, as a function of the competing cache refs/sec. The graph also shows the measured conversion rate suffered by separate functions of the MON flow.

We can see that the shape of a flow's conversion rate as a function of competition can be explained as the result of basic cache sharing. The model-derived curve has a shape similar to the empirically derived curve: sharp rise at first, significant slow-down beyond

some point. And that is even though the model provides basic probabilistic analysis of cache sharing without considering any special feature of our platform. We should note that, if we plug the model-derived conversion-rate values from Figure 4.12 into Equation 4.1 (for the value of h that corresponds to a MON flow), we get an analytical estimate of a MON flow's performance drop as a function of competition, which also has a shape similar to the corresponding empirically derived curve.

Our simple model captures the shape well, but overestimates the value of the conversion rate. This is because the model assumes that the target flow accesses its data uniformly, which is usually not the case. In Figure 4.12, we see that different MON functions are affected differently by contention:

- "flow_statistics" suffers a conversion rate that is well captured by the model, which makes sense because the flow table is accessed uniformly.

- "check_ip_header" and "skb_recycle" suffer insignificant conversion rates. Our explanation is that these functions reference the same few cacheable data with every received packet (e.g., book-keeping structures), so, their cacheable data is almost never evicted by their competitors.

- "radix_ip_lookup" is somewhere in the middle. We think this is because the root of the radix trie and its children are "hot spots," i.e., they are accessed frequently enough to remain in the cache, whereas the rest of the trie does not have any such hot spots.

However, for all functions, most of the hits that *are* susceptible to conversion are converted by the time competition reaches 50 million cache refs/sec—and our model does capture that effect.

As a side note, mathematical models that try to predict the effects of resource contention are complex because they try to analytically compute p_{ev} and p_t as a function of competition, and this is a hard task. Suppose the target flow performs r_t cache refs/sec during a solo run. The competitors cause it to suffer extra misses that "slow it down," i.e., cause it to perform fewer than r_t cache refs/sec; how much fewer depends on the competitors' cache refs/sec. At the same time, the target flow slows down the competitors by a degree that depends on the target flow's cache refs/sec. In the end, the relative frequency of target and competitor memory references (which directly affects p_t) is the result of a complex interaction among the co-runners' particular access patterns. We are able to side-step this complexity (and crudely approximate p_{ev} and p_t) because our goal is not to predict but merely to explain why increasing competition beyond some point does not significantly increase the resulting performance drop.

4.4 Predicting Resource Contention

In this section, we show how to accurately predict the overall and per-flow performance of our platform using simple profiling of each packet-processing flow running alone. Our prediction is based on the observation that a workload's aggressiveness is determined by the number of cache refs/sec that it performs, while it does not depend significantly on other workload properties.

4.4.1 Prediction Method

Suppose we plan to co-run a flow T with $|C|$ competing flows $C_1, C_2, ...C_{|C|}$. We predict flow T's performance as follows:

1. We measure the number of last-level cache refs/sec r_i performed by each flow C_i during a solo run.
2. We co-run flow T with different SYN flows, ramping up the number of cache refs/sec performed by the SYN flows, and we plot flow T's performance drop as a function of the number of competing cache refs/sec.
3. We predict that flow T's performance drop will be equal to the value of the plot (derived at step #2) that corresponds to $\sum_{i=1}^{|C|} r_i$ competing cache refs/sec.

We rely on two assumptions:

- First, we assume that the competing flows will affect the flow T as much as a set of SYN flows that perform the same number of cache refs/sec.
- Second, we assume that each competing flow C_i will perform as many cache refs/sec as it does during a solo run.

Our first assumption is well supported by the numbers in Figure 4.10. Our second assumption introduces a prediction error of a few percentage points. In reality, a competing flow C_i that belongs to a sensitive type (e.g., IP or MON) will also suffer due to contention, hence its processing will slow down, resulting in fewer cache refs/sec than it performs during a solo run. By assuming that each competing flow C_i will perform as many cache refs/sec as it does during a solo run, we overestimate the competition that flow T will encounter, hence underestimate its performance. However, the resulting prediction error is small, because of the shape of the performance drop that we observed in Section 4.3: once the number of competing cache refs/sec exceeds 50 millions or so, small changes in the number of competing cache refs/sec do not significantly change the damage inflicted to a sensitive flow. And sensitive flows like IP or MON (the ones whose

number of cache refs/sec we overestimate) are also aggressive flows, i.e., they push the number of competing refs/sec beyond the 50 million turning point.

4.4.2 Evaluation

Realistic-Flows Pairs

To validate our prediction method, we first reuse the workloads introduced in Section 4.3.1: for each possible pair of realistic flow types X and Y, we co-run a realistic flow of type X with 5 realistic flows of type Y. We have already seen the performance drop suffered by each flow type in each of these scenarios (Figure 4.2); we now look at how well we can predict these performance drops and how much of our error is due to each assumption.

Figure 4.13 shows our prediction error, i.e., the difference between predicted and actual performance drop suffered by each flow type in each scenario. For example, when a RE flow co-runs with 5 MON competitors, we predict the performance drop with an error of 3% (i.e., we predict a performance drop of 11%, while the measured performance drop is actually 8%).

Figure 4.14 shows what the error *would* be, if we knew the exact number of competing cache refs/sec—we refer to this scenario as "prediction assuming perfect knowledge of the competition". For example, when a RE flow co-runs with 5 MON competitors, if we knew the exact number of cache refs/sec performed by the 5 MON competitors, we would predict the performance drop with an error of 1.26%.

Figure 4.15 shows the absolute difference between predicted and actual performance drop suffered by each flow type, averaged across all scenarios. For example, we predict the performance drop suffered by a MON flow with an average error of 1.92% across all 5 scenarios that involve a target MON flow.

The average prediction error for each of the realistic flow types is less than 2% (the tallest bars in Figure 4.15). Our worst prediction errors are below 3%, and they correspond to the 2 leftmost bars in each group in Figure 4.13: realistic flows that co-run with 5 IP or 5 MON competitors, respectively. In these scenarios, we overestimate the performance drop suffered by the target flow, partly because we assume that its co-runners will perform as many cache refs/sec as in the solo run. Actually, IP and MON are sensitive flow types that do suffer because of contention and perform fewer cache refs/sec compared to the solo run. The difference between the corresponding bars in Figure 4.13 and Figure 4.14 represents the error introduced by our second assumption. The rest of the error is due to our first assumption that the co-runners cause as much damage as a set of SYN flows that perform the same number of cache refs/sec.

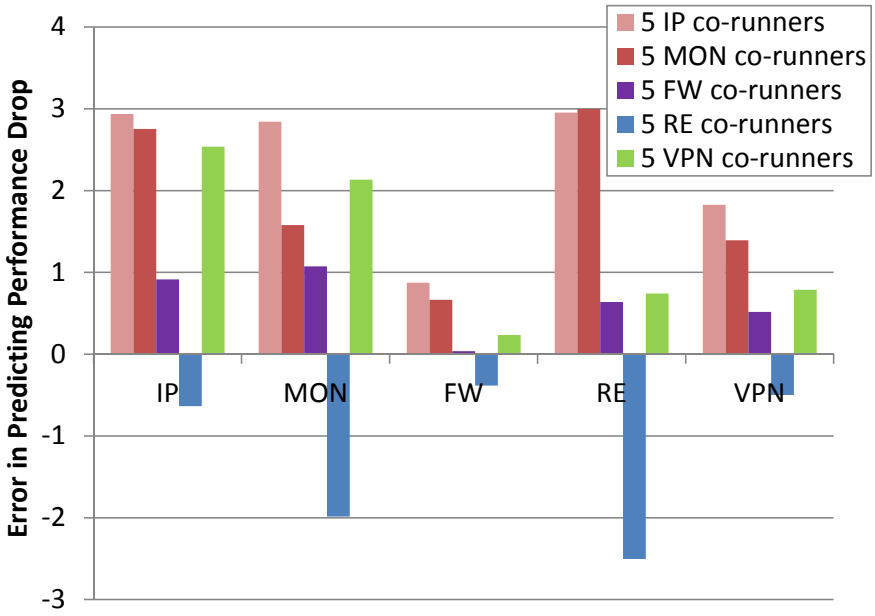

Figure 4.13: Prediction error: difference between predicted and actual performance drop suffered by each realistic flow type X when co-running with 5 realistic flows of type Y

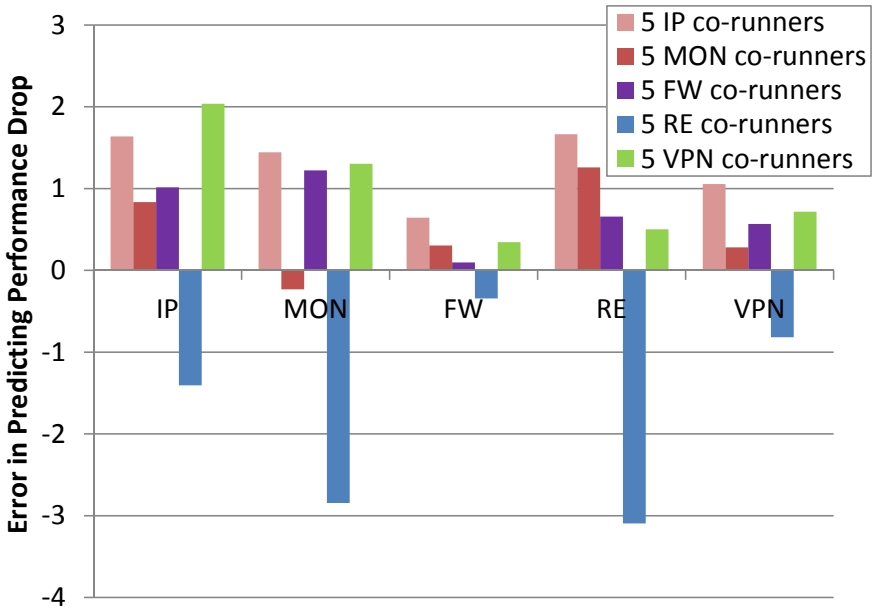

Figure 4.14: Prediction error assuming perfect knowledge of the competition: difference between predicted and actual performance drop suffered by each realistic flow type in each scenario, if we would know the exact number of cache refs/sec.

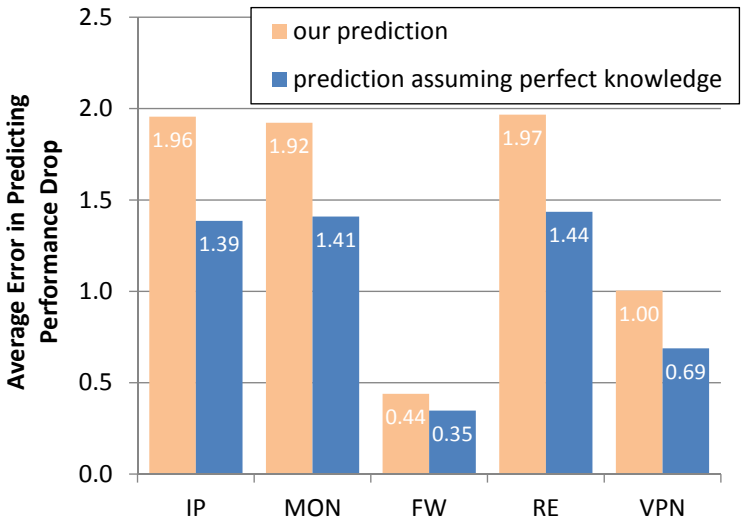

Figure 4.15: Average prediction error: average absolute difference between predicted and actual performance drop suffered by each realistic flow type across all 5 scenarios that involve a target flow of that type.

Mixed Workload

We also validate our prediction method using a mixed workload: 2 MON, 2 VPN, 1 FW, and 1 RE flow per processor. Figure 4.16 shows the actual and predicted performance drop suffered by each flow, as well as the difference between the two. This time, we predict the performance drop suffered by each flow in the mix with a maximum error of 1.26%.

4.4.3 Containing Hidden Aggressiveness

Our prediction relies on offline profiling, i.e., running each packet-processing flow alone and measuring certain properties. However, it is possible that a flow (accidentally or on purpose) exhibits different behavior during offline profiling than during the actual run—a contrived example would be a flow that normally performs FW processing (i.e., is not aggressive), but, once it receives a specially crafted packet (potentially from an attacker), it switches mode and performs SYN_MAX processing (i.e., becomes very aggressive). Such a flow could mislead the system administrator into expecting significantly higher performance from her system and under-provisioning the system accordingly.

Nevertheless, a practical implication of our results is that an administrator can control the aggressiveness of each packet-processing flow simply by throttling the flow's rate of memory accesses. To verify this, we add to the beginning of each flow a "control element," which performs a configurable number of simple CPU operations, with the purpose of

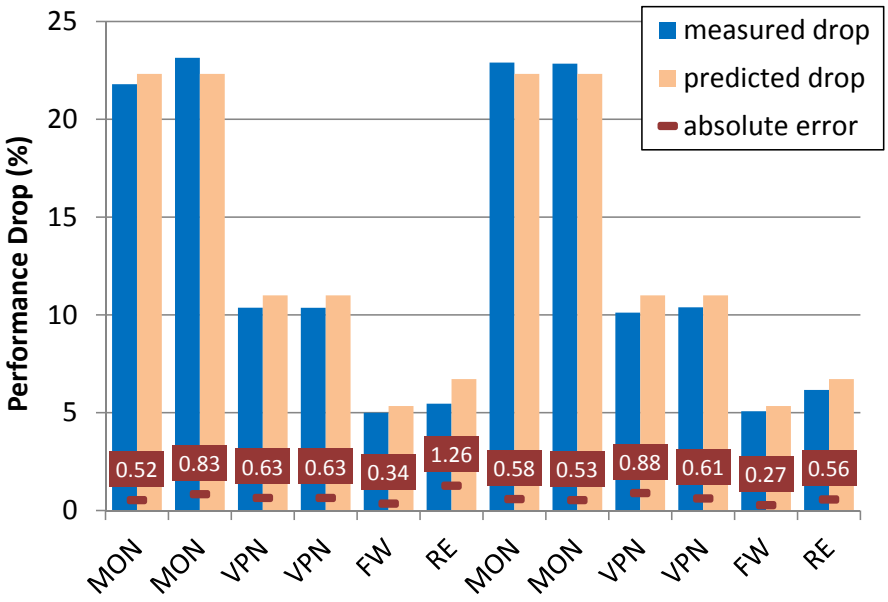

Figure 4.16: Prediction error for a mixed workload: predicted and actual performance drop suffered by each flow in the mix and the absolute difference between the two.

"slowing down" the flow and controlling the rate at which it performs memory accesses. [1] At the same time, we monitor the rate at which each flow performs memory accesses using hardware performance counters and, if a flow exceeds the rate exhibited during its offline profiling, we configure its control element to slow it down accordingly.

We tested this simple technique on our system and found that it ensures that each packet-processing flow performs no more than the profiled number of cache refs/sec. Thus, it is practical for an administrator to contain undue aggressiveness and achieve predictable performance.

4.5 Minimizing Resource Contention via Scheduling

In this section, we explore the potential benefit of contention-aware scheduling [77] for packet-processing platforms. This is a family of techniques that solve the following problem: given J processing jobs and a multicore platform with J cores, how should we assign jobs to cores to minimize resource contention between the jobs and maximize the platform's overall performance? The basic idea at the core of the proposed solutions is to profile (offline or real-time) each process and avoid co-running aggressive with sensitive processing jobs.

[1] This is better than performing explicit rate-limiting, because the latter causes the core to waste cycles polling the flow's queue for packets, only to then drop them.

To quantify the potential benefit of contention-aware scheduling for our system, we consider different combinations of 12 packet-processing flows. For each combination, we measure the contention-induced performance drop (averaged across all flows) under the worst and best flow-to-core placement (Figure 4.17). The difference between these two numbers expresses the maximum we can gain in overall system performance through contention-aware scheduling.

For realistic-flow combinations, the maximum we can gain in overall system performance is 2% (Figure 4.17). The flow combination for which we gain this maximum benefit is 6 MON and 6 FW flows.

Figure 4.18 shows the per-flow performance drop for the 6 MON and 6 FW flows combination (e.g., the leftmost set of bars corresponds to one MON flow), under the worst and best placement. The worst placement assigns the 6 MON flows to one processor and the 6 FW flows to the other, such that all the 6 MON flows (which are both aggressive and sensitive) have to compete with each other for the L3 cache; this causes a performance drop of 27% to each MON flow and an overall system performance drop of 15%. The best placement is the one that assigns 3 MON and 3 FW flows to each processor, such that each MON flow has to compete with only 2 other MON flows for the L3 cache; this causes a performance drop of 21% to each MON flow and an overall system performance drop of 13%. Hence, the extra damage introduced by the worst versus the best placement is 6% for each MON flow and 2% for the overall system.

Of all the possible realistic-flow combinations (given the flows that we implemented), this particular combination (6 MON and 6 FW flows) allows for the biggest overall improvement, because it is an equal mix of the most and least sensitive/aggressive flow types. One may think, at first, that a combination of more aggressive and/or sensitive flows (e.g., replacing the FW flows with IP or RE flows) would allow for a bigger improvement, but that is not the case. To create as big a difference as possible between the worst and best placement, we need a mix of sensitive, aggressive, and non-aggressive flows, such that in the worst placement sensitive flows co-run with the aggressive ones, whereas in the best placement sensitive flows co-run with the non-aggressive ones. Indeed, any other realistic-flow combination that we tried yielded an even smaller difference between worst and best placement.

This lack of (significant) difference between the worst and best placement can be explained using the observations in Section 4.3.3: once the competing cache refs/sec reach 50 millions or so, the performance drop suffered by a sensitive flow stays within a relatively small range, no matter which particular flows it co-runs with. Consider the 6-MON/6-FW combination: under the worst placement, each MON flow competes with 5 other MON flows, which generate about 100 million competing refs/sec, which causes the MON flow to suffer a performance drop of 27%; under the best placement, each MON flow competes with 2 other MON flows plus 3 FW flows, which generate about 60 million refs/second,

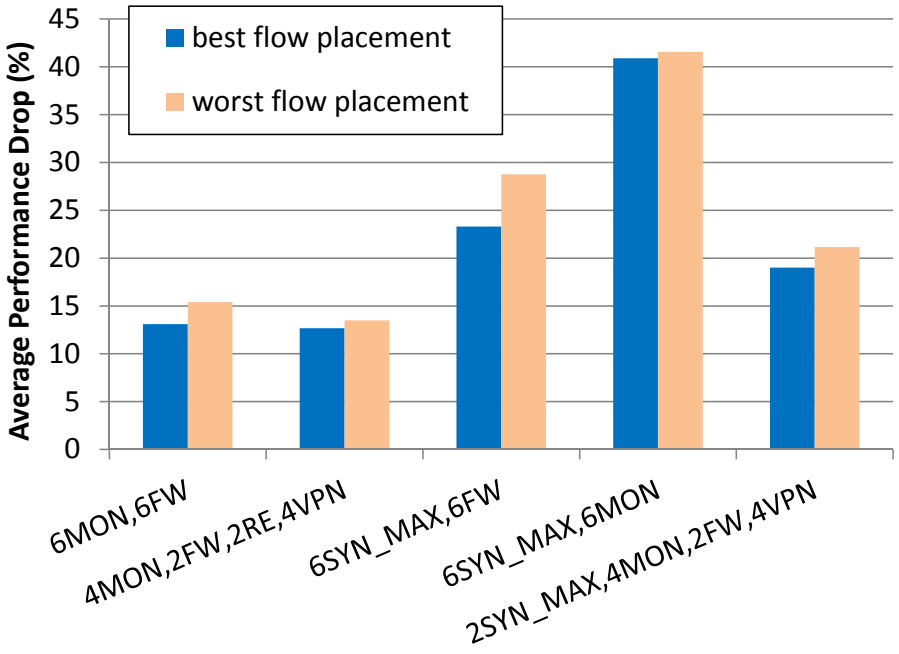

Figure 4.17: Benefit of contention-aware scheduling: average per-flow performance drop suffered under the worst and best flow-to-core placement, for different flow combinations.

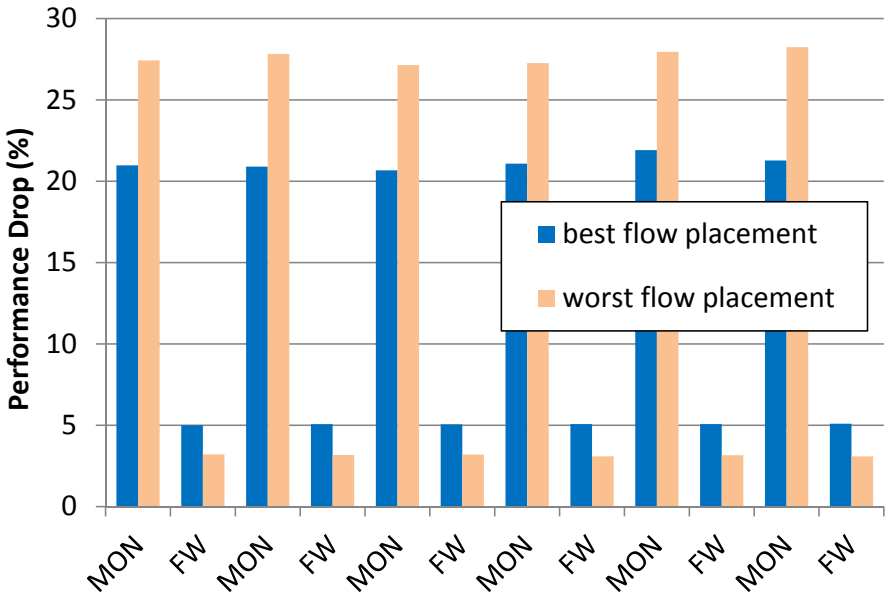

Figure 4.18: Benefit of contention-aware scheduling: per-flow performance drop suffered under the worst and best flow-to-core placement, for the 6-MON/6-FW combination.

which causes the MON flow to suffer a performance drop of 21%. In the end, as long as a placement generates more than a few tens of millions of cache refs/sec, it causes more or less the same performance drop to each sensitive flow.

If we consider non-realistic flows, the maximum we can gain in overall performance is 6%, for the 6 SYN_MAX, 6 FW combination (Figure 4.17). SYN_MAX is the most aggressive and at the same time the most sensitive flow that we were able to craft (recall that it performs no processing other than memory accesses at the highest rate possible). So, even in the scenario where we have an equal mix of flows manifesting the most aggressive/sensitive behavior that we were able to generate in our system (SYN_MAX) and non-aggressive/non-sensitive flows (FW), the maximum benefit of contention-aware scheduling with respect to overall performance is 6%. Any other combination that we tried yielded an even smaller benefit.

4.6 Discussion

All the scenarios we considered have two common characteristics: each core runs a single packet-processing flow (Section 4.2.2) and the aggregate working-set size of the competing flows far exceeds the size of the cache (Section 4.3.4).

If each core runs multiple flows, these compete for the L1 and L2 caches, so considering only the L3 accesses may not be sufficient to predict performance drop. If the working-set sizes of the flows are close to their fair share of the cache, then considering only the competing cache refs/sec may not be sufficient to characterize a workload's aggressiveness. These conditions may occur, for instance, in an active-networking setting, where large numbers of end users instantiate many small packet-processing flows on intermediate network elements.

We focused on one-flow-per-core, saturated-cache scenarios because we think that these are most likely to occur in the near future: state-of-the-art general-purpose platforms already offer tens of cores, and we consider it unlikely that a network operator would need to support more than a few tens of different packet-processing types. Moreover, the point of building programmable packet-processing platforms is to make it easy to deploy new, interesting types of packet processing. All the emerging types of packet processing that we are aware of (e.g., redundancy elimination, deep packet inspection, application acceleration) would require several megabytes of frequently accessed data in a realistic network setting (e.g., a network interface that handles a few gigabits per second, located on the border of an Internet Service Provider). In state-of-the-art platforms, the size of the shared last-level cache is less than 3MB per core (and this will not increase in the near future, if the current architecture trends persist). Hence, we expect that running any combination of interesting packet-processing applications on a state-of-the-art multicore platform would saturate the shared caches.

4.7 Related Work

In recent years, we have seen a renewed interest in general-purpose networking, both by the industry [9] and the research community. Several research prototypes have demonstrated that general-purpose hardware is capable of high-performance packet processing (line rates of 10 Gbps or more), assuming simple, uniform workloads, where all the packets are subjected to one particular type of packet processing: IP forwarding [34], GPU-aided IP forwarding [43], multi-dimensional packet classification [57], or cryptographic operations [44]. Like all this work, our ultimate goal is to build high-performance software packet-processing systems. However, our focus is to show that such a system can achieve *predictable* performance while running a wide range of packet-processing applications and serving multiple clients with different needs.

Researchers have been working for more than two decades on mathematical models for predicting the effects of resource contention. In the eighties and nineties, this was pursued in the context of general-purpose systems with simultaneous multithreading [13, 68, 73]. In the last decade, the focus has shifted to general-purpose multicore systems with shared caches [27, 30, 69, 75]. Zhang et al. recently questioned the need for prediction, with the argument that cache contention does not significantly affect the performance of modern parallel applications (in particular, PARSEC benchmarks) [76]. We show that, in the context of packet processing, resource contention can cause significant performance drop (up to 27%), however, we can accurately predict that without mathematical modeling. We should note that modeling does not remove the need for application profiling: all proposed models require as input at least the stack distance profile [59] of each application, which requires either instruction-set simulation of the application, or binary instrumentation and program analysis of the application, or co-running the application with a set of synthetic benchmarks [75].

A complementary topic to contention prediction is contention-aware scheduling: how to assign processes to cores so as to maximize overall system performance [19,33,45,51,60,77]. We show that, in the context of packet processing, contention-aware scheduling does not significantly improve overall performance.

Finally, our work falls under the broader effort of exploring how software systems should be architected to exploit multicore architectures. That work has typically focused on redesigning software to expose parallelism—most recently by eliminating serial execution bottlenecks [23]. In contrast, we focus on packet-processing workloads, which are already amenable to parallel execution. Given a seemingly perfectly parallel system like a software packet-processing platform, we analyze what are the challenges involved in running such a system and what we can do to make its performance predictable.

4.8 Summary

In this chapter, we present a software packet-processing system that combines ease of programmability with predictable performance while running a diverse set of applications and serving multiple clients with different needs. Offering predictable performance in a general context is a challenging problem, mostly because of the ways in which software processes contend for the shared hardware resources of a modern server (e.g., caches, memory controllers, buses). However, we show that in our context we can predict the effects of resource contention and achieve predictable performance.

We implement 5 forms of realistic packet processing that are deployed in the current network devices and cover a wide range of memory and CPU behavior (IP forwarding, traffic monitoring, traffic filtering, elimination of redundant traffic, traffic encryption). We configure our system to maximize its performance and avoid unnecessary contention: we parallelize workloads using the cloning approach since it results in better performance for realistic packet-processing applications; we allocate memory in a NUMA-aware fashion so that each application accesses its data locally through the memory controller that is directly connected to the processing core; we avoid forms of underlying resource contention by eliminating false sharing and unnecessary data sharing among multiple cores.

We first run a set of experiments to understand resource contention in a packet-processing system: what is the performance impact of resource contention, what is the dominant contention factor in our system and what determines an application sensitivity (i.e., the amount of damage that it suffers due to contention) or aggressiveness (i.e., the amount of damage that is causes to other applications sharing the same system). We determine that resource contention can have a significant impact on performance (e.g., an application can suffer up to 27% performance drop due to contention), while the dominant contention factor in our system is the contention for the last-level cache. We make two key observations that allow us to predict the contention-induced performance drop suffered by an application: first, we observe that performance drop of an application is mostly determined by the number of last-level cache references/second performed by its competitors, not so much by the exact type of packet processing that they perform; second, while the performance of a sensitive application drops sharply at first as the number of competing cache references/second increases, it stays within a relatively small range once the competition reaches a certain threshold.

Based on these observations, we propose and validate a prediction method which estimates the per-application performance using only simple offline profiling of each application running alone. In a nutshell, for each application sharing our system, we determine the number of cache references/second that it performs during a solo run and the performance drop as a function of the competing cache references/second when co-running with a set of synthetic applications for which we vary the level of aggressiveness (i.e., the number of

competing cache references/second). For a given application, we predict its performance drop as the value of the performance drop function corresponding to the sum of cache references/second performed by the competing applications while running solo.

Using this simple, yet accurate prediction method, we can predict the contention-induced performance drop suffered by each application in our system with an error smaller than 3%. We conclude that flexibility and predictability are not mutually exclusive goals when designing software packet-processing platforms.

We also investigate whether or not it makes sense to use contention-aware scheduling in the context of packet-processing platforms. We find that for realistic-flow combinations, the maximum we can gain in terms of overall system performance is 2%, which leads us conclude that it may not be worth the effort.

5 Dataplane Verification

Beauty is boring because it is predictable.

Umberto Eco

5.1 Introduction

Software dataplanes are emerging from both research [34, 43, 44, 57] and industry [6, 11] backgrounds as a more flexible alternative to traditional hardware switches and routers. They promise to cut network provisioning costs by half, by enabling dynamic allocation of packet-processing tasks to network devices [66]; or to turn the Internet into an evolvable architecture, by enabling continuous functionality update of devices located at strategic network points [64].

Flexibility, however, typically comes at the cost of reliability. A system of non-trivial size that is subject to frequent updates is typically plagued by behavior and performance bugs, as well as security vulnerabilities. It makes sense then that network operators are skeptical about the vision of software dataplanes that are continuously reprogrammed in response to user and operator needs—as they were skeptical a decade ago toward active networking. The question is, has anything changed? Have software verification techniques matured enough to enable us to reason about the behavior and performance of software dataplanes? Or must we accept that frequently reprogrammed software dataplanes will always be less reliable than their static hardware counterparts?

In this chapter we show a verification tool that takes as input the executable binary of a software dataplane and proves that it does (or does not) satisfy a target property; if the target property is not satisfied, the tool should provide counter-examples, i.e., packet sequences that cause the property to be violated. Developers of packet-processing apps could use such a tool to produce software with guarantees, e.g., that never seg-faults

or kernel-panics, no matter what traffic it receives. Network operators could use the tool to verify that a new packet-processing app they are considering for deployment will not destabilize their network, e.g., it will not introduce more than some known fixed amount of per-packet latency. One might even envision markets for packet-processing apps—similar to today's smartphone/tablet app markets—where network operators would shop for new code to "drop" into their network devices. The operators of such markets would need a verification tool to certify that their apps will not disrupt their customers' networks.

For general programs, verifiability and performance are competing goals. Proving properties of real programs (unlike searching for bugs) remains an elusive goal for the systems community, at least for programs that consist of more than a few hundred lines of code and are written in a low-level language like C++. A high-level language like Haskell can guarantee certain properties (like the impossibility of buffer overflow) by construction, but typically at the cost of performance.

For software dataplanes, it does not have to be this way: we argue that we can write them in a way that enables verification and preserves performance. The key question then is: what defines a "software dataplane" and how much more restricted is it than a "general program"? how much do we need to restrict our dataplane programming model so that we can reconcile verifiability with performance?

There are different ways to approach this question:

- one way is to start from a restricted, easily verifiable model and broaden it as much as possible without losing verifiability;
- another way is to start from a popular, but not verifiable model and restrict it as little as necessary to achieve verifiability.

We chose the latter in an effort to be practical. In this chapter we present the result of working iteratively on two tasks: designing a verification tool for software dataplanes, while trying to identify a minimal set of conditions that a software dataplane must meet in order to be verifiable.

We fundamentally rely on the assumption that software dataplanes follow a pipeline structure, i.e., they are composed of distinct packet-processing elements (e.g., an IP lookup element, an element that performs Network Address Translation or NAT) that are organized in a directed graph and do not share mutable state. Intuitively, the fact that there are no state interactions between elements (other than one passing a packet to another) makes it feasible to reason about each element in isolation, as opposed to having to reason about the entire pipeline as a whole. Software dataplanes that are created with Click [52] typically conform to this structure, and these arguably constitute the

majority of research prototypes. We also know of at least one industry prototype that uses Click [4], while the vision of a "composable" dataplane put forward by Intel [11] strongly implies a pipeline structure as well.

We aim to prove properties that, in the case of hardware dataplanes, are either taken for granted or can be proved using practical techniques [46–48, 58, 74]: *crash-freedom*, which means that no packet sequence can cause the dataplane to stop executing; *bounded-execution*, which means that no packet sequence can cause the execution of more than a known, reasonable number of instructions; or *filtering* properties, e.g., "any packet with source IP A and destination IP B will be dropped by the pipeline."

We describe a verification tool that proves such properties for stateless pipelines (e.g., an IP router or static firewall) and two simple stateful pipelines (a NAT box and a traffic monitor). Certain proofs assume arbitrary configuration[1], while others assume a specific one. For instance, we prove crash-freedom or bounded-execution assuming arbitrary configuration, and such proofs are useful independently of the frequency of configuration changes. In contrast, proving that a pipeline will drop a packet with given headers makes sense only given a specific configuration, and such proofs are useful when configuration changes relatively slowly.

We evaluate our tool by proving crash-freedom and bounded-execution for different Click pipelines. Our proofs complete within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within hours. Keeping verification time within minutes is necessary and sufficient given our goals: we envision our tool being used by developers, for instance to ensure that a new piece of packet-processing code cannot seg-fault, or by network operators, for instance to ensure that a given configuration change will not result in undesirable network behavior. In both cases, having to wait for hours would be impractical; waiting for tens of minutes is non-negligible, but on par with the experience of waiting for compilation to complete or configuration to be downloaded to network devices.

Even though we focus on conceptually simple pipelines, performing complete and sound verification on them required overcoming significant challenges (dealing with path explosion, loops, and large data structures). We address these challenges by applying existing verification ideas (symbolic execution [24, 38] and compositionality [14, 37, 39]) and combining them with certain domain specifics of packet-processing software (pipeline structure, bounded loops over packet contents, pre-allocated data structures that expose a key/value store interface). We share common ground with many verification tools, especially the ones that use compositional symbolic execution [14, 37, 39], but those tools were not designed with software dataplanes in mind and, and, to the best of our understanding, they cannot solve our problem.

[1] By "configuration" we mean all state that the control plane writes into the dataplane, e.g., the contents of forwarding or filtering tables.

The rest of the chapter is organized as follows: we provide the necessary background for this chapter in Section 5.2. We outline the main insight of our work in Section 5.3, we describe our system in Section 5.4 and the properties that it can prove in Section 5.5, then we evaluate our system in Section 5.6. After we discuss (potential) use cases and limitations in Section 5.7 and we present the related work in Section 5.8, then we summarize our findings in Section 5.9.

5.2 Symbolic Execution

In this section, we provide background on symbolic execution, the main verification technique that we use in our system. We provide an overview of the technique (Section 5.2.1), we discuss how we could construct proofs using symbolic execution (Section 5.2.2), what is the challenge in doing that (Section 5.2.3), and we describe S2E, a general-purpose verifier that we use in our work (Section 5.2.4).

5.2.1 Multi-Path Program Analysis

A program can be viewed as an "execution tree," where each node corresponds to a program state, and each edge is associated with a basic block. Running the program for a given input leads to the execution of a sequence of instructions that corresponds to a path through the execution tree, from the root to a leaf. For example, the program in Figure 5.1 may execute three instruction sequences: one for input $in < 0$, one for $in \geq 0 \wedge in < 10$ and another one for input $in \geq 10$. Hence, the execution tree of this program (shown to the right of the program) consists of three paths, one for each "input class" and instruction sequence.

Symbolic execution [24, 38] is a practical way of generating execution trees. During normal execution of a program, each variable is assigned a concrete value, and only a single path of the tree is executed. During symbolic execution, a variable may be symbolic, i.e., assigned a set of values that is specified by an associated constraint. For example, a symbolic integer x with associated constraint $x > 2 \wedge x < 5$ is the set of concrete values $x = \{3, 4\}$. A symbolic-execution engine can take a program, make the program's input symbolic, and execute all the paths that are feasible given this input.

Consider the program in Fig. 5.1 and assume that the input in can take *any* integer value. To symbolically execute this program, we start at the root of the tree and execute all the feasible paths. As we go down each path, we collect two pieces of information:

- the "path constraint" which specifies the values of in that lead to this path;
- the "symbolic state" which maps each variable to its current value on this path.

```
out Program ( in ):
  assert in ≥ 0
  if in < 10 then
    out ← 10
  else
    out ← in
  end if
  return out
```

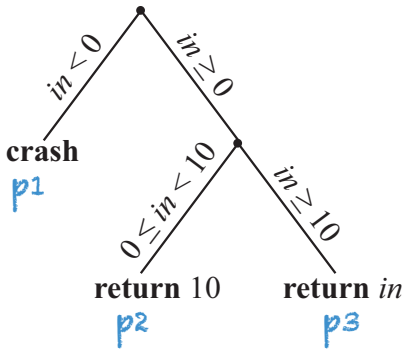

Figure 5.1: A toy program and its execution tree.

For example, at the end of path p_1, the path constraint is $C = (in < 0)$, while the symbolic state is $S = \{crash\}$; at the end of path p_2, the path constraint is $C = (in \geq 0 \wedge in < 10)$, while the symbolic state is $S = \{out \mapsto 10\}$; at the end of path p_3, the path constraint is $C = (in \geq 10)$, while the symbolic state is $S = \{out \mapsto in\}$.

5.2.2 Proof by Execution

In principle, symbolic execution provides an automatic, conceptually straightforward way to verify a program: if we can execute all the feasible paths of a program and verify that none of them violates a target property, that constitutes proof that the entire program satisfies this property. For instance, suppose we want to prove that the program in Figure 5.1 never executes more than 10 instructions. We can do this by symbolically executing the program with a symbolic input in that may take any value, executing all three feasible paths, and verifying that none of them includes more than 10 instructions.

By constructing proofs in this manner, we can also automatically determine all the problematic inputs that prevent us from completing the proof. For instance, suppose we try to prove that the program in Figure 5.1 never crashes. We symbolically execute the program with a symbolic input in that may take any value, and we discover three feasible paths: one for $in < 0$, one for $0 \leq in < 10$, and one for $in \geq 10$. We can argue that the last two paths cannot cause the program to crash, however, the first path ends with a failed assertion—a crash. So, we have failed to prove that the program satisfies the target property, but we have also uncovered all the input values ($in < 0$) that cause the property to be violated.

5.2.3 "Path Explosion" Problem

In practice, symbolic execution can be rarely used to generate proofs, because of path explosion [22]: the number of feasible paths in a program generally grows exponentially

with the number of branching points, and real software has a branching point every few instructions. As a result, the number of paths in a real program (even one that consists of a few hundred lines of code) is typically so large that it is impossible to execute all of them in useful time. For instance, even small programs like UNIX coreutils, have an intractable number of feasible paths [24].

Although symbolic execution is used successfully for identifying "good" input values for testing (e.g., values that are likely to cause bugs to manifest), it can rarely be used for proofs. The difference between testing a program and proving something about it lies in "path coverage," which is the fraction of the program's paths that are executed. Proving something about a program through symbolic execution means executing 100% of its paths, which is not a requirement for testing. A popular metric for evaluating a testing tool is "line coverage," which is the fraction of the program's lines of code that are exercised by the tool. It is worth noting that line coverage and path coverage are very different metrics; even though sophisticated tools may achieve good line coverage for certain programs, they only explore a small fraction of the feasible paths. For example, when Klee [24] tests UNIX coreutils like `nice` or `cat`, it achieves line coverage above 70%, but path coverage below 1% [54]. This is fine when the goal is to discover interesting paths in order to *uncover bugs*, but not when the goal is to reason about all feasible paths in order to *prove* properties.

Researchers have been proposing smarter ways to address path explosion [14, 37, 39, 54], but constructing complete and sound proofs for real programs that consist of more than a few hundred lines of code still requires a significant amount of manual effort [50].

Despite its limitations, we choose to build on symbolic execution for two reasons:

- First, there exist publicly available, well-maintained symbolic-execution tools [24,31] that do not require verification expertise. This is not accidental: symbolic execution can be thought of as a "brute force" approach to verification; it does not require any sophisticated reasoning about program logic, which makes it easy to use (and also hard to scale).
- Second, symbolic execution does not require access to the source code of the program, only the executable binary. This is consistent with the vision of a network operator verifying packet-processing software written by third parties, which may not have an incentive to provide the source code.

5.2.4 S2E

S2E is a state-of-the-art, publicly available verification framework for general software [31]. In our system, we use S2E to symbolically execute pieces of packet-processing code and obtain, for each piece, a set of path constraints and symbolic states.

S2E can be described as an automated path explorer with pluggable path analyzers:

- The explorer uses symbolic execution to drive the target system down multiple execution paths. S2E performs what is called "in-vivo" program analysis (as opposed to "in-vitro"), i.e., analyzes code that runs within a real software stack and this enables program analysis without having to model the underlying system (e.g., libraries, drivers, kernel, etc.).

- The analyzers measure (e.g. count the number of instructions) and/or check properties (e.g., look for bugs) of each path. User can write their own path analyzers or use one of the existing plugins that come with the S2E distribution. For instance, the "TestCaseGenerator" plugin records the set of concrete inputs for each path; the "InstructionCounter" plugin counts the number of instructions executed along each path; the "ExecutionTracer" plugin allows to rebuild the execution tree by recording the program counter whenever a fork occurs (e.g., due to branch instruction).

5.3 Insight: The Pipeline Structure

In this section, we provide an overview of our approach (Section 5.3.1), and describe our basic assumption about the structure of software dataplanes (Section 5.3.2).

5.3.1 Approach Overview

We observe that symbolic execution is a good fit for packet-processing pipelines, because their special structure can help sidestep path explosion. In a typical pipeline, two elements (stages) never concurrently hold read or write permissions to the same mutable state, regardless of whether that state is a packet being processed or some other data structure. This level of isolation can help significantly with path explosion.

Our approach is to first analyze each pipeline element in isolation, then compose the results to prove properties about the entire pipeline. This reduces by an exponential factor the amount of work that needs to be done to prove something about the pipeline: if each element has n branches and roughly 2^n paths, a pipeline of m such elements has roughly $2^{m \cdot n}$ paths. Analyzing each element in isolation—as opposed to the entire pipeline in one piece—cuts the number of paths that need to be explored roughly from $2^{m \cdot n}$ to $m \cdot 2^n$. In the worst case, the per-element analyses yield that every single pipeline path warrants further analysis—so we end up having to consider all the paths anyway. In practice, we expect that most pipeline paths are irrelevant to the target property, and we only need to consider a small fraction.

5.3.2 Packet-Processing State

We focus on packet-processing pipelines that consist of packet-processing elements, where each element may access three types of state (Table 5.1):

- *Packet state* is owned by exactly one element at any point in time. It can be read or written only by its owner; the current owner (and nobody else) may atomically transfer ownership to another element. Packet state is used for communicating packet content and metadata between elements. For each newly arrived packet, there is typically an element that reads it from the network, creates a `packet` object, and transfers object ownership to the next element in the pipeline. Once an element has transferred ownership of a `packet`, it cannot read or write it any more.
- *Private state* is owned by one element and never changes ownership. It can be read or written only by its owner, and it persists across the processing of multiple packets. A typical example is a map in a NAT element, or a flow table in a traffic-monitoring element.
- *Static state* can be read by any element, but not written by any element. This state is immutable as far as the pipeline is concerned. A typical example is an IP forwarding table.

	Written by	Read by	Transferable ownership
Packet state	owner	owner	yes
Private state	owner	owner	no
Static state	–	any	–

Table 5.1: Types of packet-processing state.

This structure is not accidental: it is a natural fit for any platform that must perform high-performance streaming. The alternative would be to allow multiple stages of the pipeline to share read/write access to the same data, which would necessarily require synchronization and the unavoidable contention and complexity that comes with it.

5.4 System Design

In this section, we describe our system: first how it leverages the pipeline structure to sidestep inter-element path explosion (Section 5.4.1); second, how it leverages other aspects of packet processing to sidestep intra-element path explosion resulting from loops (Section 5.4.2), large data structures (Section 5.4.3), and mutable state (Section 5.4.4). We close with a brief description of the kind of overhead that our design introduces (Section 5.4.5).

As we describe each technique used by our system, we also state any extra conditions on top of pipeline structure (see Section 5.3.2) that this technique requires from the target software in order to work well. If a software dataplane does not meet these conditions, our tool may not be able to complete a proof for this dataplane.

We illustrate our system through Figure 5.2, which shows a pipeline consisting of two elements. We use the term *segment* to refer to an instruction sequence through a single element, and the term *path* to refer to an instruction sequence through the entire pipeline. The input *in* corresponds to a newly received packet, and we assume that this may contain anything, i.e., we make *in* symbolic and unconstrained.

We developed our system on top of S2E [31]. For illustration purposes, our examples simplify two aspects of our system: first, our example input *in* is an integer, whereas in reality the input packet object is an array of bytes; second, our example code snippets consist of pseudo-code, whereas in reality S2E takes as input X86 code.

5.4.1 Pipeline Decomposition

The verification process in our system consists of two main steps:

- the first one searches inside each element, in isolation, for code that may violate the target property;
- the second one determines which of these potential violations are feasible once we assemble the elements into a pipeline.

More specifically, we cut each pipeline path into element-level segments (Figure 5.2). In step 1, we obtain, for each segment, a logical expression that specifies how this segment transforms state; this allows us to identify all the "suspect segments" that may cause the target property to be violated. In step 2, we determine which of the suspect segments are feasible and indeed cause the target property to be violated, once we assemble segments into paths.

In step 1, we analyze each element in isolation: first, we symbolically execute the element assuming unconstrained symbolic input. Next, we conservatively tag as "suspect" all the segments that may cause the target property to be violated. For example, in Figure 5.2, if the target property is crash-freedom, segment e_3 is tagged as suspect, because, if executed, it leads to a crash.

If we stopped at step 1, our verification would catch all property violations, but could yield false positives. If this step does not yield any suspect segments for any element, then we have proved that the pipeline satisfies the target property. For instance, if none of the elements ever crashes for any input, we have proved that the pipeline never crashes.

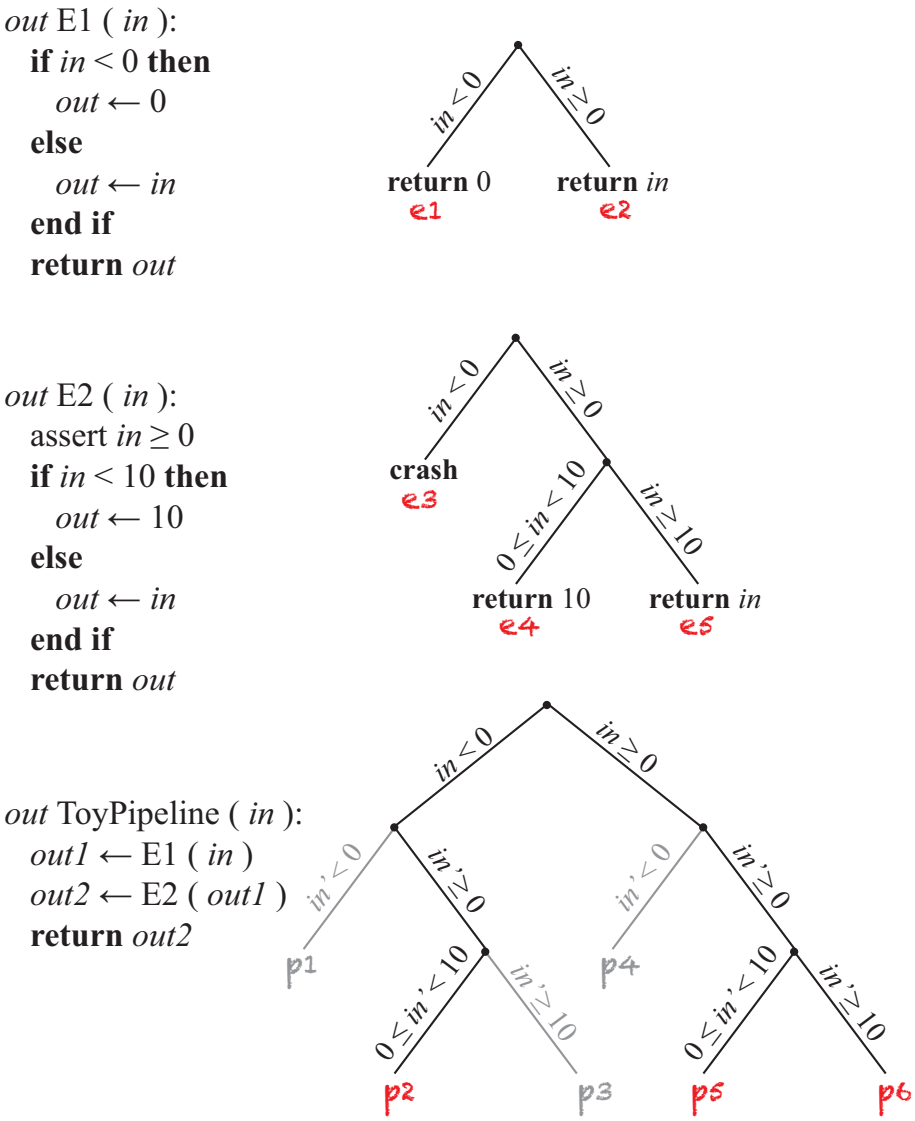

Figure 5.2: A toy pipeline that consists of two elements.

However, a suspect segment does not necessarily mean that the pipeline violates the target property, because a segment that is feasible in the context of an individual element may become infeasible in the context of the full pipeline. For example, in Figure 5.2, if we consider element E_2 alone, segment e_3 leads to a crash; however, in a pipeline where E_2 always follows E_1, segment e_3 becomes infeasible, and the pipeline never crashes. In program-analysis terminology, in step 1, we over-approximate, i.e., we execute some segments that would never be executed within the pipeline that we are aiming to verify.

Step 2 discards suspect segments that are infeasible in the context of the pipeline: first, we construct each potential path p_i that includes at least one suspect segment; each

p_i is a sequence of segments e_j. Next, we compose the path constraint and symbolic state for p_i based on the constraints and symbolic state of its constituent segments (that we have already obtained in step 1). Finally, for every p_i, we determine whether it is feasible (based on its constraints) and whether it violates the target property (based on its symbolic state). Note that the last step does not require actually executing p_i, only composing the logical expressions of its constituent segments.

For example, here is how we prove that the pipeline in Figure 5.2 does not crash:

Step 1:

1. We symbolically execute E_1 assuming input *in* can take any integer value. We collect the following constraints and symbolic state for its segments e_1 and e_2.
   - $C_1(in) = (in < 0)$, $S_1(in) = \{out = 0\}$.
   - $C_2(in) = (in \geq 0)$, $S_2(in) = \{out = in\}$.
2. We symbolically execute E_2 assuming input *in* can take any integer value. We collect the following constraints and symbolic state for its segments e_3, e_4, and e_5:
   - $C_3(in) = (in < 0)$, $S_3(in) = \{crash\}$.
   - $C_4(in) = (in \geq 0 \wedge in < 10)$,
     $S_4(in) = \{out = 10\}$.
   - $C_5(in) = (in \geq 10)$, $S_5(in) = \{out = in\}$.
3. We tag segment e_3 as suspect.

Step 2:

1. The paths that include the suspect segment are p_1 (i.e., sequence $< e_1, e_3 >$) and p_4 (i.e., sequence $< e_2, e_3 >$).
2. We compute p_1's path constraint as $C_1^*(in) =$

   $$C_1(in) \wedge C_3(\ S_1(in)\ [out]\ ) =$$
   $$C_1(in) \wedge C_3(\ 0\ ) = (in < 0) \wedge (0 < 0) = False.$$

3. We compute p_4's path constraint as $C_4^*(in) =$

   $$C_2(in) \wedge C_3(\ S_2(in)\ [out]\ ) =$$
   $$C_2(in) \wedge C_3(\ in\ ) = (in \geq 0) \wedge (in < 0) = False.$$

4. Both path p_1's and path p_4's constraints always evaluate to false, hence p_1 and p_4 are infeasible, i.e., there are no feasible paths that include suspect segments, hence the platform never crashes.

Pipeline decomposition enables us to prove properties about the pipeline without having to consider every single pipeline path; but it still requires us to consider every single element segment. This is not straightforward for elements that involve loops, large data structures, and/or mutable private state. We will next discuss how we address each of these scenarios.

5.4.2 Loops

In general, loops can be a challenge for program verification, especially when the number of loop iterations depends on the input. For example, a loop of t iterations, where t is a 64-bit unsigned integer input, can yield 2^{64} execution paths.

In contrast to general programs, a software dataplane typically will not contain input-dependent loops with such a large number of maximum iterations. A worst-case realistic example is a packet-processing element that loops over the bytes of a packet for encryption or compression. In this case, the number of loop iterations is bounded by the maximum packet size, typically 1500.

Still, loops can create an impractical number of segments within an element. Consider an element that implements the processing of IP options: for each received packet, it loops over the options stored in the packet's IP header and performs the processing required by each specified option type. If the processing of one option yields up to 2^n segments, then the processing of t options yields up to $2^{t \cdot n}$ segments. For example, in the IP-options element that comes with the Click distribution, the processing of 3 options yields millions of segments that—we estimated—would take months to symbolically execute.

To address this, we reuse the idea of decomposition, this time applying it not to the entire pipeline, but to each loop: if a loop has t iterations, we view it as a "mini-pipeline" that consists of t "mini-elements", each one corresponding to one iteration of the loop. We have described how, if we have a pipeline of m elements, we symbolically execute each element in isolation, then compose the results to reason about the entire pipeline. Similarly, if we have a loop of t mini-elements (iterations), we symbolically execute each mini-element in isolation, then compose the results to reason about the entire loop. Unlike a pipeline that consists of different element types, a loop of t iterations consists of the same mini-element type, repeated t times; hence, for each loop, we only need to symbolically execute one mini-element.

This brings us to our first extra condition on packet-processing code: to use decomposition as we do, the only mutable state shared across components must be the packet object

itself. For instance, to decompose a pipeline into individual elements, we rely on the fact that the only mutable state shared across elements is the packet object. Similarly, to decompose a loop into individual iterations, the only mutable state shared across iterations must be part of the packet object.

For example, consider again an IP-options element: such an element typically includes a next variable, which points to the IP-header location that stores the next option to be processed; each iteration of the main loop starts by reading this variable and ends by incrementing it. In a conventional element, next would be a local variable. In our verification-optimized element, next is part of the packet metadata, hence part of packet. And since, in step 1, we make packet symbolic and unconstrained, next is also symbolic and unconstrained, allowing us to reason about the behavior of one iteration of the main loop, assuming that iteration may start reading from *anywhere* in the IP header.

Condition 1 *Any mutable state shared across loop iterations is part of the packet metadata.*

To make a packet-processing element satisfy this condition, a developer needs to identify any variables that are read and written across loop iterations and make these variables part of the packet metadata. For the Click IP-options element, this process required changing 26 lines (12%) of the code and took less than an hour. Alternatively, this can be done automatically by a compiler (that would force developers to explicitly declare mutable state shared across iterations of a loop). Either way, this condition does not restrict the functionality that a packet-processing element can implement; it only forces the developer to create—either manually or with compiler help—an explicit interface between loop iterations.

5.4.3 Data Structures

Symbolic-execution engines lack the semantics to reason about data structures in a scalable manner. For instance, symbolically executing an element that uses a packet's destination IP address to index an array with a thousand entries will cause a symbolic-execution engine to essentially branch into a thousand different segments—independently from the array content or the logic of the code that uses the returned value. So, if we naïvely feed an element with a forwarding or filtering table of more than a few hundred entries to a symbolic-execution engine, step 1 of our verification process will not complete in useful time.

To address this, when we reason about an element, we abstract away any data-structure access. This allows us to symbolically execute the element and identify suspect segments, without requiring the symbolic-execution engine to handle any data structures. To reason about the data structures themselves, we rely on other means, e.g., manual or static

analysis; this restricts us to using only data structures that are manually or statically verifiable, but we have evidence that these are typically sufficient for packet-processing functionality.

This brings us to our second extra condition on packet-processing code: to reason about different components of the same executable separately, there must exist a well-defined interface between them. For instance, to reason about each pipeline element separately and compose the results, we rely on the existence of a well-defined interface between each pair of elements, which specifies all the state that can be exchanged between them (the packet object). Similarly, to reason about a data structure separately from the element that uses it and compose the results, the data structure must expose a well-defined interface to the element.

We need an interface that abstracts a data structure as a key/value store that supports at least read, write, membership test, and expiration. The first three operations are straightforward; the last one—expiration—allows an element to indicate that a {key, value} pair will not be accessed by the element anymore, hence is ready to be removed and processed by the higher layers. For example, suppose an element maintains a data structure with per-flow packet counters; when a flow completes (e.g., because a FIN packet from that flow is observed), the element can use the expiration operation to signal this completion to the control-plane process that manages traffic statistics.

Condition 2 *Elements use data structures that expose a key/value-store interface like the one in Figure 5.3.*

```
value          =  read ( key )
                  write ( key, value )
{True, False}  =  test ( key )
                  expire ( key, value )
```

Figure 5.3: An interface for dataplane data structures.

Moreover, we need data structures that expose the above interface *and* can be verified in useful time. When we say that a data structure is "verified", we mean that the implementation of the interface exposed by the data structure is proved to satisfy crash-freedom, bounded-execution, and correctness. The latter depends on the particular semantics of the data structure, e.g., a hash-table should satisfy the following property: a "write (key, value)" followed by a "read (key)" should return "value." If an element uses only data structures for which these properties hold, then, when we reason about the element, we can abstract away all the data-structure implementations and consider only the rest of the element code plus the data-structure interfaces.

Condition 3 *Elements use data structures that are implemented on top of verifiable building blocks, e.g., pre-allocated arrays.*

As evidence that such data structures exist, we implemented a hash table and a longest-prefix-match table that satisfy crash-freedom and bounded-execution. They both consist of chains of pre-allocated arrays. For instance, our hash table is a sequence of K such arrays; when adding the k-th key/value pair that hashes to the same index, if $k \leq K$, the new pair is stored in the k-th array, otherwise it cannot be added (the write operation returns False). For the longest-prefix-match table, we use the idea of "flattening" of all entries to /24 prefixes [42].

We chose arrays as the main building block, because they combine two desirable properties: (i) they enable line-rate access to packet-processing state, because of their $O(1)$ lookup time; and (ii) they are easy to verify, because of the simplicity of their semantics. For example, a write to an array (that is within the array bounds) is guaranteed not to cause a crash and not to cause the execution of more than a known number of instructions that depends on the particular CPU architecture. In contrast, a write to a dynamically-growing data structure, e.g., a linked list or a radix trie, may result in a variable number of memory allocations, deallocations and accesses, which can fail in unpredictable ways.

5.4.4 Mutable Private State

Mutable private state is hard to reason about because it may depend on a sequence of observed packets (as opposed to the currently observed packet alone). For instance, if an element maintains connection state or traffic statistics, then its private state is a function of all traffic observed since the element was initialized. Hence, it is not enough to reason about the segments of the element that can result from all possible contents of the current packet; we need to reason about the segments that can result from all possible contents of all possible packet sequences that can be observed by the element. The challenge is that symbolic-execution engines (and verification tools in general) are not yet at the point where they can handle symbolic inputs of arbitrary length in a scalable manner.

We have only scratched the surface of verifying elements with mutable state. We can currently verify two kinds of elements that maintain mutable state: a Network Address Translator (NAT) that maintains per-connection state and rewrites packet headers accordingly, and a traffic monitor that maintains per-flow packet counters. We believe that our approach can be generalized to other elements, but we do not expect to be able to perform complete and sound verification of an element that performs arbitrary state manipulation—claiming that would be close to claiming that we could verify arbitrary software.

Our approach is to break verification step 1 (Section 5.4.1) into two sub-steps: the first one searches for "suspect" values of the private state that would cause the target property to be violated, while the second one determines which of these potential violations are

feasible given the logic of the element. In the first sub-step, we assume that the private state can take *any* value allowed by its type (i.e., we over-approximate). In the second sub-step, we take into account the fact that private state cannot, in reality, take any value, but is restricted by the particular type of state manipulation performed by the given element. So far, we have not needed to exercise the second sub-step in practice: in the two stateful elements that we have experimented with, the first sub-step did not reveal any suspect states, hence the second one was not exercised. We describe both sub-steps through a manufactured example in Appendix B.

5.4.5 Overhead

Our extra conditions introduce two kinds of overhead:

- First, existing elements may need to be rewritten to satisfy them. This involves rewriting existing loops, replacing existing data structures, and changing any line of code that accesses a data structure.

  To reduce this overhead, part of our work is to create a library of data structures that satisfy our conditions.

- Second, implementing sophisticated data structures on top of pre-allocated arrays typically requires more memory than conventional implementations.

  In our opinion, sacrificing memory for verifiability is worth considering given the relative costs of memory and the human support for dealing with network problems.

We put specific numbers of these kinds of overhead in Section 5.6, where we present our evaluation results.

5.5 Target Properties

In this section, we describe the three target properties that our prototype can prove or disprove.

5.5.1 Crash-Freedom

We say that a pipeline is *crash-free* when it is guaranteed not to execute any instruction that would cause it to terminate abnormally, e.g., an assertion with a false argument or a division by zero. The definition of "abnormal termination" depends on the environment where the pipeline runs: in the case of user-mode Click, it is the receipt of a signal (e.g., SIGSEGV, SIGABRT, SIGFPE) that is not handled by the Click process and causes the process to terminate; in the case of kernel-mode Click, it is a call to the kernel's panic

method. As stated in Section 5.4, our tool is built on top of an in-vivo path explorer, which can detect any of these conditions.

We prove crash-freedom for a pipeline given an arbitrary input packet and arbitrary configuration state. If a pipeline does not include any instruction that may cause abnormal termination, proving crash-freedom is trivial. On the other hand, if a pipeline does include an instruction that may cause abnormal termination, that does not necessarily mean that this instruction may be executed. So, proving crash-freedom is equivalent to proving that any such instruction will never be executed, and proving lack of crash-freedom is equivalent to providing a specific packet and specific state that causes such an instruction to be executed.

5.5.2 Bounded-Execution

We say that a pipeline satisfies *bounded-execution* when it is guaranteed to execute no more than I_{max} instructions per packet. This ensures that no packet is ever caught in an infinite loop. It can also be used to produce a "latency envelope," i.e., argue that once a packet enters the pipeline, it will exit within a bounded amount of time. To translate instruction sequences into latency bounds, we need to map each instruction to the minimum and maximum number of cycles that it can take to complete, which can be typically obtained from the CPU and/or chip manual.

We prove bounded-execution for a pipeline given an arbitrary input packet and arbitrary configuration state. We find the longest path of a pipeline as follows:

- In step 1 of the verification process, when we symbolically execute an element, we also record the length (number of instructions) of each of its segments.
- In step 2, we search for the longest feasible path by considering different segment combinations.

We use a simple search heuristic that first checks if the path that consists of the longest segment of each element is feasible (if yes, we are done), then checks if any path that involves either the first or second longest segment of each element is feasible, and so on. Although, in a worst case scenario, we might have to check all possible segment combinations, in practice, we find the longest feasible path after considering only a few combinations.

5.5.3 Filtering

Given a pipeline with specific configuration state, can we guarantee that a packet that enters the pipeline with source IP A and destination IP B will be dropped?

We leverage existing work that answers this type of question for hardware dataplanes [46–48, 58, 74]. This work abstracts each network device as a function that maps an input packet header to an output port, and then it composes different device functions to reason about the entire network; the mapping function of each device is determined by the contents of its forwarding table. In contrast, we abstract each packet-processing element as a function that maps an input packet header to an output port, and then we compose different element functions to reason about the entire pipeline; the mapping function of each element is automatically derived by symbolically executing the element's code given an arbitrary input packet.

The main difference lies in the derivation of the mapping function of each packet-processing element (that we do by symbolically executing the element in isolation). This is useful in cases where an element, e.g., includes a line of code that drops all packets with source IP A, even though the device's forwarding table indicates otherwise. Composing element functions to reason about a pipeline is equivalent to composing device functions to reason about a network, and we can reuse the algorithms proposed by the above work.

5.6 Evaluation

We tested our ideas on pipelines created with Click. In each tested pipeline, packets are generated by a "generator" element and dropped by a "sink" element; what we verify is the packet-processing code between generator and sink.

We answer the following questions:

- Can we perform complete and sound verification of software dataplanes (Section 5.6.1)?
- How does verification time increase with pipeline length (Section 5.6.2)?
- Can we use our tool to uncover bugs, useful performance characteristics, or unintended dataplane behavior (Section 5.6.3)?

5.6.1 Feasibility and Overhead

We considered pipelines with three types of elements: (1) Elements from Click release 2.0.1 that we did not change at all: Classifier, CheckIPHeader, EtherEncap, DecIPTTL, DropBroadcasts, and Strip (Ethernet decapsulation); (2) Elements from the same release that we modified modestly: IPGWOptions and DirectIPLookup; (3) Elements that we wrote from scratch: a Network Address Translator (NAT) and a traffic monitor that maintains per-flow packet counters.

We verified various pipelines that consist of combinations of these elements. For each

pipeline, we proved crash-freedom and bounded-execution. More generally, for each pipeline, we were able to answer questions of the following kind: can line X in element Y be executed with arguments Z in the context of this pipeline? If yes, what is a packet that would cause this line to be executed with these arguments?

In Table 5.2 we indicate the origin of each element: whether it is an original element from Click release 2.0.1 ("Click"), one that we modified ("Click+"), or one that we wrote from scratch ("ours"). The table also indicates the number of lines of code that we modified or introduced in each element ("New lines of code") and which of our techniques were needed to complete step 1 of the verification process for each element (marked with "X"s).

Element	New lines of code (% of total)	Loops	Data Structures	Mutable State
Click:				
Classifier				
CheckIPHeader				
EtherEncap				
Strip				
DecIPTTL				
DropBroadcasts				
Click+:				
IPGWOptions	26 (12%)	X		
DirectIPLookup	130 (20%)		X	
Ours:				
NAT	870		X	X
TrafficMonitor	650		X	X

Table 5.2: Verified packet-processing elements.

We have not yet applied our approach widely enough to have statistically meaningful results on the amount of rewriting effort. To make the IPGWOptions and DirectIPLookup elements satisfy our conditions, we had to change, respectively, about 26 lines of code (12% of the element's code) and 130 lines of code (20% of the element's code), which took a few hours. Our modifications consisted of loop rewriting and replacing data structures with our verifiable ones. We also tried to make the Click IPRewriter element from release 2.0.1 (which generalizes NAT functionality) satisfy our conditions, but we ended up changing most of the element, because most of its code accesses data structures. In the end, we wrote it from scratch (about 870 lines of code), which took a couple of days. Our traffic monitor is about 650 lines of code.

In terms of memory overhead, the Click IPRewriter element stores per-connection state in a hash table, implemented as an array of dynamically growing linked lists (when adding the k-th key/value pair that hashes to the same index, the new pair is stored as the k-th item of a linked list). In contrast, our NAT element uses the hash-table

implemented as a chain of $K = 3$ pre-allocated arrays (this value makes the probability of dropping a connection negligible). Hence, our NAT element may use up to 3 times more memory to store the same amount of state.

5.6.2 Scalability

We now examine how verification time increases with pipeline length. Given the intended uses of our tool, it should not take more than a few tens of minutes to prove a target property per pipeline. We first look at meaningful pipelines (that it makes sense to actually deploy), then at microbenchmarks that illustrate different aspects of our system. To show the benefit of our domain-specific techniques, we use as a baseline vanilla S2E. We refer to our tool as "dataplane-specific verification" and to S2E as "generic verification". We feed the same code to the two systems.

Meaningful Pipelines

We consider three meaningful pipelines:

(a) *edge router* implements a standard IP router (the first 8 elements in Table 5.2) with a small forwarding table (10 entries);

(b) *core router* is similar but has a large forwarding table (100,000 entries);

(c) *network gateway* implements NAT and per-flow statistics collection.

Each of them presents an extra verification challenge: the first one includes a loop (processing IP options), the second one a large data structure (IP lookup table), and the third one mutable private state (our NAT and TrafficMonitor elements).

In Figure 5.4 we show how verification time increases as we add more elements to the IP-router pipelines: dataplane-specific verification completes in less than 20 minutes. Most of this time is spent on IP options, because this element has significantly more branching points than the rest. Generic verification of the edge router exceeds 12 hours (at which point we abort it) the moment we allow packets to carry 2 IP options (so we do not show any data point for it beyond "+IPoption2"). Generic verification of the core router exceeds 12 hours the moment we add the IP lookup element to the pipeline (so we do not show any data point for it beyond "+IPlookup"). The difference between the two tools comes from our pipeline decomposition and the special treatment of loops and large data structures.

In Figure 5.5 we show the same information for the network-gateway pipeline: dataplane-specific verification completes in less than 6 minutes, whereas generic verification exceeds

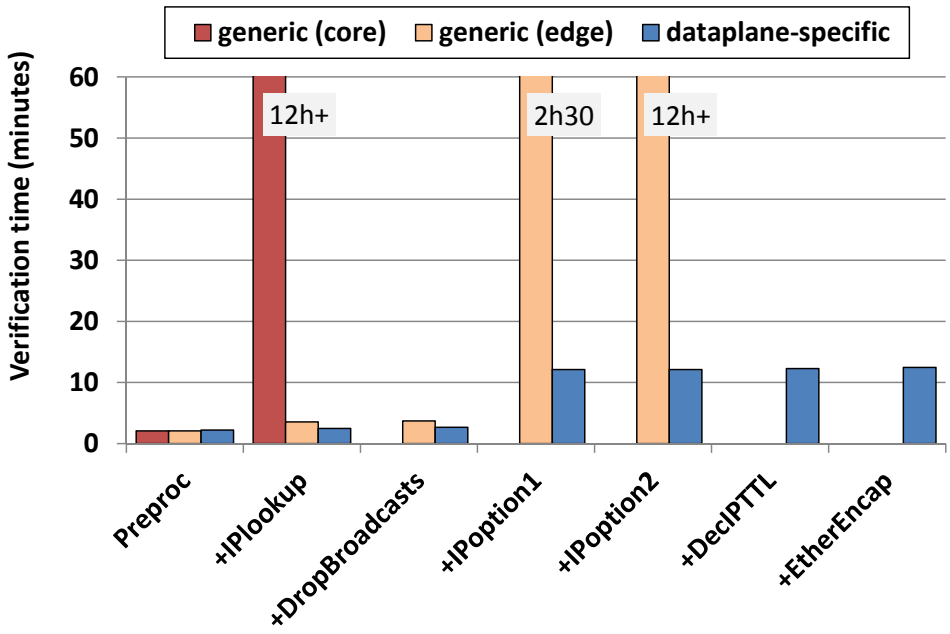

Figure 5.4: IP router. Verification time as a function of pipeline length. For the dataplane-specific tool, the results are the same for the edge and core pipelines.

12 hours the moment we add either the TrafficMonitor or the NAT element. The difference comes from our pipeline decomposition and the fact that we abstract away data-structure implementations.

Compositionality Microbenchmarks

We now consider two synthetic pipelines to illustrate the benefit of pipeline and loop decomposition:

(a) The first one consists of a sequence of simple filtering elements, each of which reads a different part of the input packet's IP header to make a filtering decision.

(b) The second one implements a simplified version of the IP options processing loop, i.e., in each iteration, it reads some portion of the IP header, updates it, and advances a `next` variable that indicates where the next read should start.

In Figure 5.6 we show how verification time increases as we add more filtering elements to the first pipeline: generic verification time increases significantly faster than dataplane-specific verification time. This is because the former executes all feasible segments of each element in isolation, whereas the latter executes all feasible paths of the pipeline.

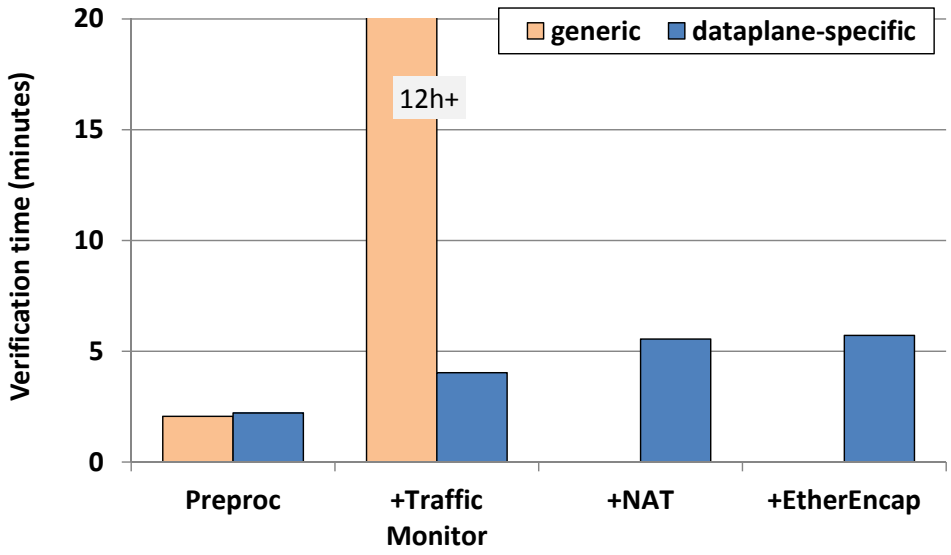

Figure 5.5: Network gateway. Verification time as a function of pipeline length.

In this scenario, generic verification does complete in useful time, because this pipeline involves few elements, without loops, that access minimal state. Still, it takes an order of magnitude more time than dataplane-specific verification because of the exponential increase in the number of paths. To make this clear, we note, on top of each bar, the number of verification states that each tool generates and processes.

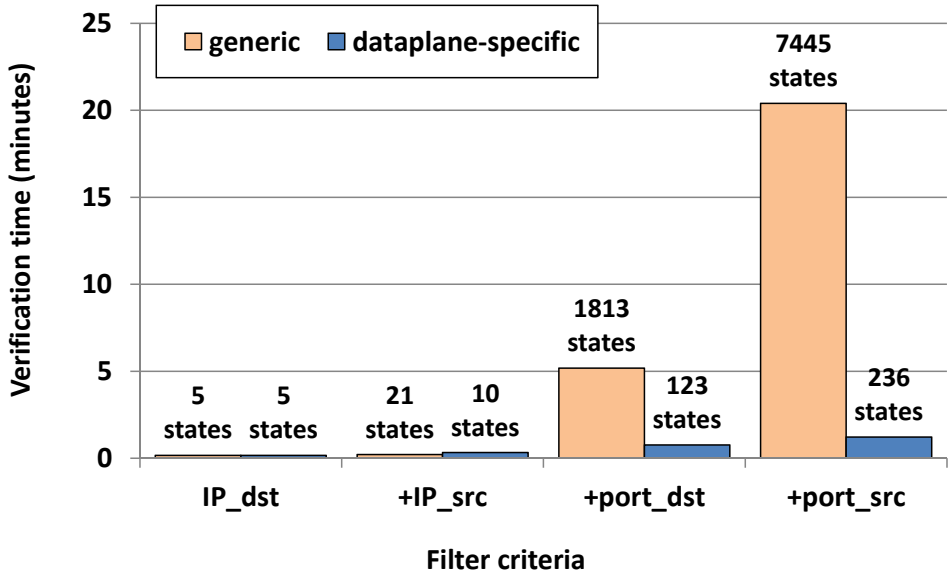

Figure 5.6: Pipeline microbenchmark. Verification time as a function of pipeline length.

In Figure 5.7 we show how verification time increases as we add more iterations to the loop of the second pipeline: dataplane-specific verification time remains constant, whereas generic verification time increases exponentially. This is because the former executes all feasible segments of one loop iteration, whereas the latter executes all feasible paths of the entire loop. Dataplane-specific verification is slower than generic verification only in the special case where we have a loop with a single iteration. That is because it symbolically executes one loop iteration, assuming that iteration may start reading from *anywhere* in the IP header; this pays off as soon as we add a second loop iteration, but it is unnecessary in the special case of a loop with a single iteration.

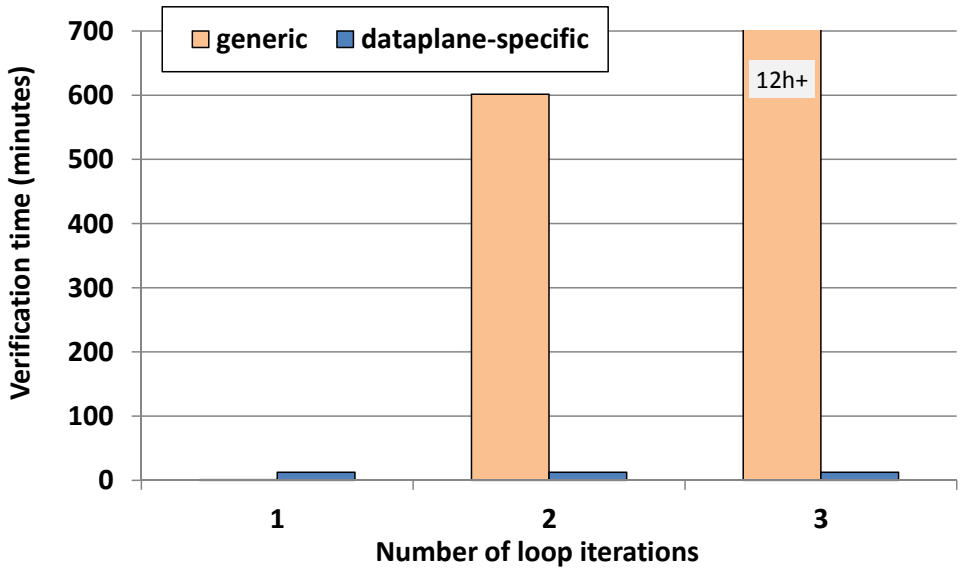

Figure 5.7: Loop microbenchmark. Verification time as a function of pipeline length.

5.6.3 Usefulness

We said that our tool can help developers debug their code, and network operators better understand the performance and behavior of their dataplanes; we now look at a few specific examples.

Bugs in Existing Code

While trying to prove crash-freedom and bounded-execution for various pipelines, we encountered the following instances where a malicious end-host could disable the pipeline by sending a specially crafted packet. All elements that we refer to belong to Click release 2.0.1 (without any subsequent patches).

Bug #1: Any pipeline that includes the IPFragmenter element will enter an infinite loop, if it tries to fragment a packet with IP options. This is because the for loop that processes IP options in the fragmenter does not have an increment (the programmer forgot to add one).[2]

Bug #2: Any pipeline that includes the IPFragmenter element, not preceded by the IPGWOptions element, will enter an infinite loop if it tries to fragment a packet that carries a zero-length IP option. This is because the current option length determines where the next iteration of the loop will start reading, hence, a zero-length option causes the loop to get stuck.[3] The IPGWOptions element discards any packet with a zero-length option, so including it in the pipeline prevents the bug from being exercised.

Bug #3: Any pipeline that includes the IPRewriter element[4] will hit a failed assertion[5] if it receives a packet with source IP address/port tuple $T_s = T$ and destination tuple $T_d = T$, where T is the public IP address/port of the IPRewriter. We should note that this particular bug was fixed (independently from us) in a patch subsequent to release 2.0.1.

How hard would it be to find these bugs manually? The first one is probably not that hard: a loop missing its increment stands out visually, plus any serious testing of the fragmenter element would involve a packet with IP options. The other two bugs, however, manifest in scenarios that a developer is unlikely to test, but an attacker can easily exploit: fragmentation of an illegal packet while processing of IP options is turned off (which does happen, in practice, for performance or security reasons); and processing of an IP header that would be meaningless in a legitimate packet.

In Section 5.3.1, we said that we expected verification step 2 to compose the constraints only for a small fraction of the pipeline paths. Table 5.3 reports, for a given bug and pipeline, the amount of time spent and the number of paths composed in this step. Consider bug #2: when verifying a pipeline that includes the Click IP fragmenter element, step 1 determines that this element has a suspect segment. If the pipeline does not support IP options, step 2 determines that the suspect segment is feasible in this pipeline (hence the pipeline does not satisfy bounded-execution); this requires finding *one* feasible path that contains the suspect segment, and we succeed after composing the constraints for 26 paths, which takes 5 seconds. If the pipeline supports one IP option, step 2 determines that the suspect element is not feasible in this pipeline; this requires showing that *all* the paths that contain the suspect segment are not feasible, and we succeed after composing the constraints for 8423 paths, which takes 47 minutes. These numbers are consistent with our expectation that, in practice, verification step 2 completes in useful time.

[2] elements/ip/ipfragmenter.cc, line 64
[3] elements/ip/ipfragmenter.cc, line 69
[4] elements/tcpudp/iprewriter.cc
[5] include/click/heap.hh, line 149

Bug	Pipeline	Time	# Paths
#1	Edge router with 1 IP option + Click IP fragmenter	3 min	432
#2	Edge router with 1 IP option + Click IP fragmenter	47 min	8423
#2	Edge router without options + Click IP fragmenter	5 sec	26
#3	Network gateway with Click NAT	5 sec	10

Table 5.3: Time spent and number of paths composed in verification step 2, when the pipeline contains buggy elements.

Longest Paths in IP Router

We used our tool to construct adversarial—from a performance point of view—workloads for a pipeline implementing a standard IP router. Recent research showed that such a router is capable of multi-Gbps lines rates [34], but this result was obtained using workloads of well-formed packets, not meant to exercise the pipeline's exception paths. Instead, we obtained the pipeline's 10 (it could have been any number) longest paths, as well as the packets that cause them to be executed.

It is not surprising that the longest paths are executed in response to problematic packets that trigger further packet examination and logging; what may be surprising is that these paths execute 2.5 times as many instructions than the most common path. Moreover, these extra instructions are CPU-heavy, i.e., they include memory accesses and system calls for logging; an attacker may cause significant performance degradation by sending a sequence of packets that are specially crafted to exercise these particular paths. This is useful information to a developer, because it reveals to him paths that may require his attention. It is also useful to a network operator, because it reveals to her the performance limits of a pipeline and the workloads that trigger them—allowing her to decide whether it is suitable for her network.

Unintended Behavior

Certain implementations of the Loose Source Record Route (LSRR) IP option may enable illegal traffic to bypass a firewall [40]: an IP router that supports the LSRR option may replace the source IP address of an incoming packet with its own IP address. In this case, any filtering based on the source IP address of the packet that happens *after* the processing of IP options is ineffective. This has been exploited to bypass firewalls, eventually causing network operators to disable LSRR and router manufacturers to change their LSRR implementations.

Our tool would have uncovered this vulnerability. To verify that, we created a pipeline that includes an IP options element followed by a firewall, and we tried to prove that it satisfies the following property: "any packet whose source IP address is blacklisted by the firewall will be dropped." The tool responded that the property is not satisfied, and it provided an example packet that causes it to be violated: a packet with a blacklisted source IP address that carries the LSRR option.

5.7 Discussion

In this section, we discuss the potential use cases of a verification tool for software dataplanes (Section 5.7.1), the limitations of our system (Section 5.7.2), and our plan towards addressing them (Section 5.7.3).

5.7.1 Use Cases

The most obvious users of a verification tool for software dataplanes could be the *developers of packet-processing code*. Reasoning about the behavior of a packet-processing element E is hard enough; reasoning about what E will do when part of a bigger pipeline is even harder. Our tool would help by checking, for any given design or implementation choice in E, what would be the impact on one or more bigger pipelines that include E. It would also provide concrete examples of packets that lead to a segmentation fault, a division by 0, or a failed assertion. Today, developers are forced to perform extensive testing before release and our tool would make them more productive by focusing their attention on the most relevant test cases.

A second set of users could be *network operators*: when a new, interesting type of packet processing becomes available (e.g., a new form of intrusion detection or application acceleration), an operator may want to include this as a new element E in the pipelines running on its network devices. Today, the operator has no effective way of assessing the consequences of such an upgrade on the network as a whole. At best, it can test for a while and deploy widely after gaining some level of confidence that there will be no dire consequences. As a result, trying out new packet-processing software is time-consuming and potentially dangerous. A dataplane verification tool would change this by providing a way to check what would be the impact on the currently running pipelines (e.g., what would be the maximum increase in latency or energy consumption that the new element would introduce). Such information would enable faster and safer deployment, ultimately making operators less conservative in trying out new packet-processing software.

A third—and perhaps most interesting—use case targets future *markets for packet-processing elements* that are similar to today's app markets like Apple AppStore or Google Play. Such markets would allow network operators to "go shopping" for new

packet-processing elements that they can then drop into the dataplanes of their network devices. Our tool could help by enabling the app-market operator to formally certify that the desired element will not disrupt the customer's pipeline.

5.7.2 Limitations

The key enabler and at the same time limitation of our work is that we focus on software dataplanes that are subject to certain restrictions:

1. They obey a pipeline structure (Section 5.3.2). We think that this is a natural fit for packet-processing software. Most research prototypes are already written this way, and we know of at least one industry prototype as well. Favoring an already popular programming model is, in our opinion, a modest price to pay for verifiability.
2. They satisfy our extra conditions (Section 5.4), which may require rewriting existing code (but the resulting code is, in our opinion, easier to read and maintain); and may result in implementations with larger memory footprints than existing code (but trading off memory for verifiability is, in our opinion, worth considering).

We currently handle only two specific, simple forms of mutable private state. As stated earlier, we do not expect to be able to completely remove this limitation, but we do expect to expand the range of state-manipulation patterns that we can formally reason about.

Our approach is applicable to packet-processing platforms where each packet is handled by a single processing core and different cores never need to synchronize. We focused on such platforms, because there is compelling evidence that they lead to better performance by minimizing the number of compulsory cache misses (Chapter 3). We would need new results in order to verify platforms where different cores contend for access to the same data structures.

5.7.3 Future Work

As we continue our work, we are coming to the following conclusion: symbolic execution is a powerful tool, but it alone cannot take us to complete and sound verification of sophisticated, stateful packet-processing elements. We were able to use it thus far, because we considered only stateless elements (IP router) or elements that maintain relatively simple forms of state (NAT and traffic monitor). To verify stateful elements, we first assume that the element's state may take any value and check whether the target

property holds; if it does not hold for a particular suspect value, we then try to reason about whether the element's state can ever take that suspect value (Section 5.4.4). The challenge lies in the second step. In the case of our NAT and traffic monitor, there were no suspect values to begin with—and even if there were, reasoning about the feasible values of the state of these elements is relatively simple. However, an intrusion detection system (IDS) may implement sophisticated state machines that we cannot reason about, even with our domain-specific optimizations.

Hence, we are heading toward an approach where we analyze each element in isolation and compose the results on demand (as our current system does), but we use a wider range of techniques for the per-element analysis. Techniques like model checking or abstract interpretation may be necessary if we want to reason about sophisticated, stateful packet-processing elements. The challenge lies in simplifying these techniques enough to make them accessible to non-verification experts; continuing in the same vein with our current work, we will try to do this by leveraging the idiosyncrasies of packet-processing code. For example, we are exploring the idea of identifying patterns of code that are common in packet processing, and developing specialized techniques for verifying these particular patterns.

There will always be software that we cannot verify, and we by no means advocate that network operators reject it. Rather, we advocate classifying software into "verified" and "non-verified," to enable network operators to make informed decisions about deploying it on their network devices. For instance, it makes sense to deploy non-verified software in an isolated fashion (e.g., within a container, where only experimental traffic can reach it), to contain its effect on the rest of the network. Perhaps this combination of verification and dual-mode deployment is the way to reconcile the need for predictable network behavior and performance with the vision of a continuously evolving Active Network.

5.8 Related Work

Our work is feasible because of advances in program analysis tools for C/C++ code, from Verisoft [36] to modern model checkers [53, 63] and tools based on symbolic execution [24, 25, 31, 38]. These tools target general code, so they cannot typically construct complete and sound proofs (which is what we want). Instead, they try to increase line coverage or identify buggy paths *without* having to reason about all the paths of the analyzed program (whereas we want to reason about *all* feasible paths of the analyzed pipeline). There exist tools that prove properties of real programs, but, to the best of our knowledge, they are tailored to specific domains other than dataplanes (e.g., avionics, device drivers); notable examples are Astrée [20], SLAM [16, 17], and Terminator [32].

Compositionality has been leveraged before to address path explosion, in compositional dynamic test generation [37] and follow-on work [14, 39]. The particular tools evaluated

in these proposals use "top-down" composition: when symbolically executing a program, encountering a function triggers the construction of a summary (logical representation) of that function in the context of its caller. This makes sense, because—to the best of our understanding—the goal of this line of work is to maximize line coverage with as little work as possible (ideally, hit each program statement exactly once). We use "bottom-up" composition: we first compute context-free summaries of all elements, then we compose them as necessary to reason about the entire pipeline.

Verification techniques have been used before to debug or verify networked systems (but not dataplanes): Musuvathi and Engler adapted the CMC model checker to test the Linux TCP implementation for interoperability with the TCP specification [62]. Bishop et al. contributed a formal specification of TCP/IP and the sockets API, and they tested existing implementations for conformance to their specification [18]. Killian et al. contributed new algorithms for finding liveness bugs in systems like Pastry and Chord [49]. NICE finds bugs in OpenFlow applications [26]. SOFT tests OpenFlow switches for interoperability with reference implementations [55]. Guha et al. contributed "the first machine-verified [Software Defined Networking] controller" [41].

Ennals et al. contributed a new language for packet-processing applications [35]. The goal of that language was to simplify the "compilation of high-level programs to the distributed memory architectures of modern Network Processors." The proposed language ensured that no two threads referenced the same packet, which is akin to our requirement that no two pipeline elements have access to the same packet.

5.9 Summary

In this chapter, we explore the feasibility of verifying software dataplanes to ensure smooth network operation. While in the context of general-purpose software verifiability and performance are competing goals, we show that software dataplanes are different and they can be written in a way that enables verification while preserving performance.

We describe a verification tool that takes as input the executable binary of the software dataplane and proves that it does (or does not) satisfy target properties like crash-freedom (i.e., no packet can cause the dataplane to stop executing), bounded-execution (i.e., no packet can cause the execution of more than a known, reasonable number of instructions) and filtering properties (e.g., the dataplane will filter all packets with source IP A and destination IP B).

Our work relies on the key observation that software dataplanes typically follow a pipeline structure, and they can be written as a set of distinct packet-processing elements that are organized in a directed graph and do not share mutable state. In particular, we focus on packet-processing pipelines containing elements that may access three types of

state: packet state (owned by exactly one element at any point in time), private state (owned by one element and never changes ownership) and static state (can be read by any element and can not be written by any element).

Our approach is to first analyze each pipeline element in isolation and look for code that may violate a target property, then compose the results to prove properties about the entire pipeline. This reduces by an exponential factor the amount of work that needs to be done to prove something about the pipeline. However, even reasoning about a single element in isolation poses significant challenges when dealing with elements that involve loops and data structures. We identify a set of extra conditions (besides the pipeline structure) that the code needs to satisfy in order to be verification friendly. For instance, any mutable state shared across loop iterations should be part of the packet itself (i.e. packet object metadata), elements should use data structures that expose a key/value-store interface and are implemented on top of verifiable building blocks (e.g., pre-allocated arrays).

We evaluate our work using Click pipelines. We are able to perform complete and sound verification of stateless pipelines and two simple stateful pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within several hours.

6 Conclusions

Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.

Albert Einstein

Software packet-processing devices represent an appealing alternative to the hardware switches and routers because they are easy to program and, therefore, they enable the fast deployment of new, sophisticated kinds of packet processing without the need to buy and deploy expensive new equipment. They could take us to a world of evolvable networks that can adapt better to the needs of the network operators and provide improved services for the users.

However, a flexible system (of non-trivial size) that gets frequently reprogrammed brings along the risk of unpredictable performance and behavior. Consider, for instance, a new piece of code that gets deployed on a software network device and, upon receiving a specially crafted packet, it enters an infinite loop which causes the network device to stop processing packets; or a new type of intrusion detection that, once it gets deployed, it causes an unpredictable drop in performance for other applications sharing the same hardware.

The challenge is to offer flexibility while, at the same time, achieve a competitive level of performance and predictability. Previous projects have demonstrated software platforms that are capable of high performance, but only for simple, conventional workloads like packet forwarding and IP routing and only after careful manual calibration. Moreover, this was achieved under the simplifying assumption of uniformity: all processing cores process the same type and amount of traffic and run identical code. Instead, in this dissertation, we focus on systems that support a wide range of sophisticated packet-processing applications and we show that it is feasible to build packet-processing platforms that are flexible and predictable, while preserving performance and ease of programming.

First, we look at how we should parallelize a data-flow functionality across the available processing cores in order to maximize the system's throughput. We understand what parallelization options are possible—we defined these as cloning vs. pipelining—, then we identify three distinguishing overheads (i.e., inter-core synchronization, cache contention for data structures, workload imbalance) that suggest the existence of a trade-off between the two approaches. After extensive experiments quantifying these overheads, we conclude that, in practice, there is no real trade-off and the cloning approach is always better.

Second, we present a software packet-processing system that combines ease of programmability with predictable performance, while running a diverse set of applications and serving multiple clients with different needs. Despite the unpredictable ways in which applications contend for shared hardware resources like caches, memory controllers and buses, we can perform accurate prediction of the effects. This is the result of two key observations: i) the performance drop suffered by a given flow due to contention is mostly determined by the number of cache references per second performed by its competitors and ii) as long as the number of competing cache references per second exceeds a certain threshold, the performance drop stays within a relatively small range for sensitive flows (i.e., those that suffer the most). We show that, in our system, we can predict the performance drop suffered by each flow due to contention with an error smaller than 3% and the overall system performance depends little on how we schedule different flows on different cores, hence, contention-aware scheduling may not be worth the effort.

Third, we present a verification tool that takes as input a software dataplane, written in a way that meets a given set of conditions, and proves that it either does or does not satisfy useful properties (e.g., crash-freedom) in order to ensure smooth network operation. Our main insight is that packet-processing is particularly suitable for software verification because of the limited state interactions between different stages of the processing pipeline (i.e., they communicate with each other through a well-defined, narrow interface and do not share mutable state). This makes it possible to reason about the behavior of the entire dataplane without treating it as a single piece of code—we can reason about each processing element in isolation and then compose the results to reason about the entire dataplane. We combine this observation with existing techniques from software verification (e.g., symbolic execution and compositionality) and we build a system that is able to perform complete and sound verification of stateless and two simple stateful Click pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within several hours.

We consider these results to be good news for all the ongoing efforts in general-purpose networking: we conclude that when designing software packet-processing platforms, flexibility does not have to come at the cost of predictability.

A A Simple Cache Model

We consider a target flow T and a competitor C sharing a direct-mapped cache of C cache lines. The target flow achieves H_t hits/sec during a solo run and accesses W chunks of cacheable data (where a chunk has the size of a cache line). The competitor performs R_c references/sec during a solo run.

We make the following simplifying assumptions:

1. The competing flow accesses the cache uniformly, i.e., when it makes a memory reference, that maps to a particular cache line with probability $\frac{1}{C}$.
2. The target flow accesses its cacheable data uniformly, i.e., when it references its cacheable data, it references a particular chunk with probability $\frac{1}{W}$.
3. The target flow and the competing flow are equally sensitive to cache contention, i.e., they suffer the same performance drop.

These assumptions differ from reality in different degrees, depending on the nature of the target and of the competing flow.

We can express the target flow's hit-to-miss conversion rate as follows:

- Consider a sequence of cache references, $\langle c_1, c_2, ..., c_Z, t' \rangle$, where: t and t' are two consecutive references performed by the target flow to the same chunk, t' was a hit during a solo run, and $c_i, i = 1..Z$, are the competing references that occur between t and t'.
- Suppose that each competing reference c_i evicts the content cached by t with probability p_{ev}, independently from any other competing reference. t' is a hit if none of the Z competing references evict this content, i.e.,

$$P(hit|Z) = (1 - p_{ev})^Z. \tag{A.1}$$

- Suppose that each reference that occurs after t is: either a competing reference, with probability p_c, or t', with probability $p_t = 1 - p_c$. Hence, Z is a random variable of geometric distribution with success probability p_t, i.e.,

$$P(Z = z) = (1 - p_t)^z p_t. \tag{A.2}$$

- By combining Equation A.1 and Equation A.2,

$$\begin{aligned} P(hit) &= \sum_{z=0}^{\inf} P(hit|Z)P(Z = z) \\ &= \sum_{z=0}^{\inf} (1 - p_{ev})^z (1 - p_t)^z p_t + \\ &= \frac{p_t}{1 - (1 - p_{ev})(1 - p_t)}. \end{aligned} \tag{A.3}$$

- The target flow's hit-to-miss conversion rate is $1 - P(hit)$.

Next, we estimate p_{ev} and p_t:

- Based on assumption #1,

$$p_{ev} = \frac{1}{C}. \tag{A.4}$$

  This is because each competing reference hits the same cache line as t with this probability.

- Based on assumptions #2 and #3,

$$p_t = \frac{H_t \cdot \frac{1}{W}}{H_t \cdot \frac{1}{W} + R_c}. \tag{A.5}$$

  Under a solo run, the competing flow performs R_c references/sec, while the target flow performs $H_t \cdot \frac{1}{W}$ references/sec to each of its chunks that are hits. Assuming that the target and competing flows suffer the same performance drop, the ratio between competing references and target references to one particular chunk remains constant during the run, which yields the above equation.

By substituting Equation A.4 and A.5 into Equation A.3, we get an estimate of $P(hit)$ and the target flow's hit-to-miss conversion rate.

We used this model to derive the theoretical (gray) curve shown in Figure 4.12, which represents an estimate of the hit-to-miss conversion rate suffered by a MON flow when it competes with 5 synthetic flows, as a function of the competing references/sec. As discussed in Section 4.3.4, this estimate is not accurate enough to be used for prediction (it is only accurate for the flow_statistics function of the MON flow, because that function accesses each flow-table entry with the same probability). We use it only to demonstrate that the shape of the hit-to-miss conversion rate as a function of competition can be explained as the result of basic cache sharing.

B Mutable Private State Example

Reasoning about elements that maintain mutable private state is hard because the state may depend not on a single packet, but on a sequence of packets (i.e., the private state is a function of all traffic observed since the element was initialized). We illustrate this using the code example in Figure B.1 where an element maintains a private data structure (*map*) with per-flow packet counters and we want to prove that a packet counter never overflows (i.e., the target property).

```
1: if map.exists(flowId) = false then
2:     map.write(flowId, 0)
3: end if
4: pktCnt ← map.read(flowId)
5: newPktCnt ← pktCnt + 1
6: map.write(flowId, newPktCnt)
```

Figure B.1: Code that collects per-flow packet counters.

The element performs the following operations:

- Lines 1–3 add a new entry for the current flow (where the current packet belongs), if such an entry does not already exist.
- Line 4 reads the counter of the current flow.
- Line 5 increments the variable that stores the read.
- Line 6 updates the counter.

We said that our approach is to break verification step 1 (Section 5.4.1) into two sub-steps (Section 5.4.4): we first assume that the private state can take any value allowed by its type, then we restrict the possible values of the private state to those allowed by state manipulation performed by the element.

More concretely, in the first sub-step (i), we identify and tag as "suspect" all the values of the private state that would cause the target property to be violated. To achieve this,

we make symbolic and unconstrained not only the input packet and its metadata, but also any variable that stores a read from a private data structure. In Figure B.1, there is one such variable: *pktCnt*. Making *pktCnt* symbolic and unconstrained allows us to identify the segments that may result from all the *pktCnt* values allowed by *pktCnt*'s type.

The output of the symbolic-execution engine is:

$$C_1(in, pktCnt) = (), \qquad \text{(B.1)}$$
$$S_1(in, pktCnt) = \{newPktCnt \mapsto pktCnt + 1\}.$$

This indicates that, if the current flow counter is equal to max (the maximum value allowed by its type), then the counter will overflow.

If we stopped at the first sub-step, our verification would be complete but not sound: a suspect value does not necessarily mean that the element violates the target property, because the logic of the element may preclude some of these values.

In the second sub-step (ii) we make the verification sound, by checking which suspect values are actually feasible given the state transformation performed by the element (Equation B.1). In our example, we use the output of the first sub-step (i) to prove by induction that max is a feasible value for *pktCnt*, and *newPktCnt* will overflow, as long as the element observes a sequence of max +1 packets.

As described so far, our approach for dealing with mutable private state is semi-manual: we said that we use the output of the first sub-step (i) to prove by induction that *newPktCnt* can overflow; we construct this proof manually, as we are not aware of any practical, publicly available tool that will do it for us automatically. However, semi-manual verification is not practical—we do not envision software engineers writing formal proofs.

To remove the need for manual construction of proofs, we could turn the second sub-step (ii) into a pattern-matching problem:

1. Identify patterns of symbolic state that occur frequently in packet-processing elements.
2. Manually construct proofs about the state contained in these specific patterns.
3. When reasoning about an element, to prove that a suspect value of the private state is or is not feasible, try to match the symbolic state that corresponds to the private state to one of the common patterns for which we have pre-constructed proofs. For instance, if an element contains a piece of private state that matches the pattern in Equation B.1, then this element contains a counter that will eventually overflow.

Bibliography

[1] Cisco Carrier Routing System. http://cisco.com/en/US/products/ps5763/index.html.

[2] Cisco IOS NetFlow. http://www.cisco.com/web/go/netflow.

[3] Intel 82599 10 GbE Controller Datasheet. http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[4] Meraki. http://meraki.cisco.com.

[5] OProfile. http://oprofile.sourceforge.net.

[6] Vyatta Hardware Appliances. http://www.vyatta.com/solutions/physical/appliances.

[7] Vyatta Series 2500. http://vyatta.com/downloads/datasheets/vyatta_2500_datasheet.pdf.

[8] Vyatta Series 3500. http://vyatta.com/downloads/datasheets/vyatta_3500_datasheet.pdf.

[9] Why Use Vyatta? http://www.vyatta.org/getting-started/why-use.

[10] Cars and Software Bugs. http://www.economist.com/blogs/babbage/2010/05/techview_cars_and_software_bugs, 2010.

[11] Intel RFP Announcement: SDN Extensions for Programmable Data Services, 2012.

[12] Boeing 787 Software Bug. http://www.theregister.co.uk/2015/05/01/787_software_bug_can_shut_down_planes_generators, 2015.

[13] A. Agarwal, M. Horowitz, and J. Hennesy. An Analytical Cache Model. *Transactions on Computer Systems (TOCS)*, 7:184–215, 1989.

[14] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[15] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can Software Routers Scale? In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2008.

[16] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. of the ACM EuroSys Conference*, 2006.

[17] T. Ball and S. K. Rajamani. SLAM: Debugging System Software via Static Analysis. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2002.

[18] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *Proc. of the ACM SIGCOMM Conference*, 2005.

[19] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-Aware Contention Management on Multicore Processors. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[20] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[21] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[22] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[23] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[24] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] C. Cadar and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM Conference on Computer Communication Security (CCS)*, 2006.

[26] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[27] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[28] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proc. of the USENIX Security Symposium*, 2011.

[29] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *Proc. of the USENIX Annual Technical Conference*, 2001.

[30] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[31] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1), 2012.

[32] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[33] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy-Efficient Computing in Virtualized Environments. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[34] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[35] R. Ennals, R. Sharp, and A. Mycroft. Linear Types for Packet Processing. In *European Symposium on Programming*, 2004.

[36] P. Godefroid. Model Checking for Programming Languages Using Verisoft. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 1997.

[37] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2007.

[38] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[39] P. Godefroid, A. Nori, S. Rajamani, and S. D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2010.

[40] F. Gont, R. Atkinson, and C. Pignataro. Recommendations on Filtering of IPv4 packets Containing IPv4 Options. http://tools.ietf.org/html/draft-ietf-opsec-ip-options-filtering-05#section-4.3.

[41] A. Guha, M. Reitblatt, and N. Foster. Machine-Verified Network Controllers. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[42] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proc. of the IEEE INFOCOM Conference*, 1998.

[43] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proc. of the ACM SIGCOMM Conference*, 2010.

[44] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[45] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[46] P. Kazemian, M. Chang, H. Zeng, S. Whyte, G. Varghese, and N. McKeown. Real Time Network Policy Checking using Header Space Analysis. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[47] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[48] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[49] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[51] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28:54–66, 2008.

[52] E. Kohler, R. Morris, B. Chen, J. Jannoti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, August 2000.

[53] D. Kroening, E. Clarke, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[54] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[55] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proc. of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[56] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[57] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. of the ACM SIGMETRICS Conference*, 2010.

[58] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proc. of the ACM SIGCOMM Conference*, 2011.

[59] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9:78–17, 1970.

[60] A. Merkel, J. Stoess, and F. Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the EuroSys Conference*, 2010.

[61] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems (TOCS)*, 15(3):217–252, Aug. 1997.

[62] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[63] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[64] B. Raghavan, T. Koponen, A. Ghodsi, M. Casado, S. Ratnasamy, and S. Shenker. Software Defined Internet Architecture. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[65] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. of the USENIX Annual Technical Conference*, 2012.

[66] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[67] N. T. Spring and D. Wetherall. A Protocol Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the ACM SIGCOMM Conference*, 2000.

[68] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2005.

[69] D. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[70] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[71] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, (1), January 1997.

[72] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *ACM Computer Communication Review (CCR)*, (2), April 1996.

[73] D. Thiebaud and H. S. Stone. Footprints in the Cache. *Transactions on Computer Systems (TOCS)*, 5:305–329, 1987.

[74] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static reachability Analysis of IP Networks. In *Proc. of the IEEE INFOCOM Conference*, 2005.

[75] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[76] E. Z. Zhang, Y. Jiang, and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Comtemporary Multithreaded Programs? In *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[77] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Mihai Dobrescu

Network Architecture Lab (NAL) & Operating Systems Lab (LABOS)
EPFL, Switzerland – Computer and Communication Sciences

Email: mihai.dobrescu@epfl.ch
Web: http://people.epfl.ch/mihai.dobrescu

Research Interest

Network systems with emphasis on software packet-processing systems

Education

2009 – present Network Architecture Lab & Operating Systems Lab, EPFL, Switzerland
PhD student in Computer and Communication Sciences

2007 – 2009 Politehnica University, Bucharest, Romania
Master of Science in Systems Programming and Applications

2002 – 2007 Politehnica University, Bucharest, Romania
Bachelor of Science in Computer Science and Engineering

Honors & Awards

- Best paper award NSDI 2014 and SOSP 2009
- Travel grant award SOSP 2011, SOSP 2009 and PACT 2007

Publications

2014 Software Dataplane Verification, Mihai Dobrescu and Katerina Argyraki, USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2014.

2013 Toward a Verifiable Software Dataplane, Mihai Dobrescu and Katerina Argyraki, ACM Workshop on Hot Topics in Networks (HotNets), November 2013.

2012 Toward Predictable Performance in Software Packet-Processing Platforms, Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy, USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2012.

2011 RouteBricks: enabling general purpose network infrastructure, Kevin Fall, Gianluca Iannaccone, Maziar Manesh, Sylvia Ratnasamy, Katerina Argyraki, Mihai Dobrescu, Norbert Egi, in Operating Systems Review, January 2011.

2010	Controlling Parallelism in Multi-core Software Routers, Mihai Dobrescu, Katerina Argyraki, Maziar Manesh, Gianluca Iannaccone, and Sylvia Ratnasamy, in the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO), November 2010.
2010	Evaluating the Suitability of Server Network Cards for Software Routers, Maziar Manesh, Katerina Argyraki, Mihai Dobrescu, Norbert Egi, Kevin Fall, Gianluca Iannaccone, Eddie Kohler, and Sylvia Ratnasamy, in the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO), November 2010.
2009	RouteBricks: Exploiting Parallelism to Scale Software Routers, Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy, in the ACM Symposium on Operating Systems Principles (SOSP), October 2009.

Work and Research Experience

2009 – 2015	EPFL, Switzerland - research assistant working on designing predictable software packet-processing systems
Summer 2009, 2010	Intel Research Labs, Berkeley CA, USA - visiting intern working on designing and analyzing a scalable software router.
2006 – 2009	Politehnica University, Bucharest, Romania - teaching staff working on preparing lectures, labs, exam tests for undergraduate classes.
Spring, Summer 2008	EPFL, Switzerland - research intern working on designing and analyzing a scalable software router.
Spring, Summer 2007	University of Toronto, Canada - research intern working on designing and analyzing a distributed software transactional memory system.

Teaching

Fall 2012, 2013, 2014	Computer Networks, EPFL, Switzerland
Fall 2006, 2007, 2008	Local Area Networks, Politehnica University, Bucharest, Romania
Fall 2008	Operating Systems Introduction, Politehnica University, Bucharest, Romania

Academic Service

External reviewer	Transactions on Networking 2013, INFOCOM 2013, Information Processing Letters 2013, USENIX ATC 2012, CCR 2012