DISS.ETH NO. 23474

Rethinking host network stack architecture using a dataflow modeling approach

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Pravin Shinde
Master of Science, Vrije Universiteit, Amsterdam
born on 15.10.1982
citizen of Republic of India

accepted on the recommendation of

Prof. Dr. Timothy Roscoe, examiner
Prof. Dr. Gustavo Alonso, co-examiner
Dr. Kornilios Kourtis, co-examiner
Dr. Andrew Moore, co-examiner

2016

Abstract

As the gap between the speed of networks and processor cores increases, the software alone will not be able to handle all incoming data without additional assistance from the hardware. The network interface controllers (NICs) evolve and add supporting features which could help the system increase its scalability with respect to incoming packets, provide Quality of Service (QoS) guarantees and reduce the CPU load. However, modern operating systems are ill suited to both efficiently exploit and effectively manage the hardware resources of state-of-the-art NICs. The main problem is the layered architecture of the network stack and the rigid interfaces.

This dissertation argues that in order to effectively use the diverse and complex NIC hardware features, we need (i) a hardware agnostic representation of the packet processing capabilities of the NICs, and (ii) a flexible interface to share this information with different layers of the network stack.

This work presents the Dataflow graph based model to capture both the hardware capabilities for packet processing of the NIC and the state of the network stack, in order to enable automated reasoning about the NIC features in a hardware-agnostic way. It also describes the implementation of the Dragonet network stack which uses the dataflow model to support the diversity and complexity in the NICs.

Dragonet enables effective use of the available hardware features in a portable way by using the dataflow model to share the packet processing capabilities of a particular NIC configuration with the runtime system, which can then automatically specialize the network stack based on this information.

Furthermore, the dataflow model enables systematic exploration of the hardware configuration space and allows reasoning about the hardware capabilities in the context of application requirements. Thus, allowing policy-based management of the NIC's resources.

The dissertation shows the effectiveness of the dataflow model by implementing several high-level policies for managing different hardware resources on two separate NICs.

Kurzfassung

Mit der wachsenden Diskrepanz zwischen der Geschwindigkeit des Netzwerks und der CPU-Leistung ist Software nicht mehr in der Lage alle eingehenden Daten ohne jegliche Hilfe von der Hardware zu bewältigen. Netzwerk Interface Controller (NIC) entwickeln sich daher ständig weiter und können mittlerweile den Systemen helfen, höhere Skalierbarkeit bezüglich eingehender Pakete zu erreichen, Quality-of-Service (QoS) Garantien bereitzustellen oder die CPU-Last zu reduzieren. Modernen Betriebssystemen fehlt allerdings die Unterstützung um diese Erweiterungen moderner NICs effizient auszunutzen und verwalten zu können. Das Hauptproblem in diesem Zusammenhang ist die Architektur des Netzwerkstapels in Schichten sowie zu inflexible Schnittstellen.

Diese Dissertation argumentiert, dass für die effektive Nutzung vielfältiger und komplexer NIC-Hardware sowohl eine Repräsentation der Paketverarbeitung existieren muss, die unabhängig von der verwendeten Hardware ist, als auch eine flexible Schnittstelle um diese Information in allen Schichten des Netzwerkstapels verfügbar zu machen.

Diese Arbeit präsentiert ein auf einem Dataflow-Graph basierende Modell, welches sowohl die Fähigkeiten der Hardware im Bezug auf die Paketverarbeitung eines NIC als auch den Status des Netzwerkstapels darstellt. Dies erlaubt die automatische, hardware-unabhängige Auswahl der verfügbaren Fähigkeiten der NIC-Hardware. Des Weiteren beschreibt die Arbeit die Implementierung des Dragonet Netzwerkstapels, welcher das Dataflow-Modell benutzt um die Vielfalt und Komplexität der NICs optimal zu unterstützen.

Dragonet erlaubt mit Hilfe des Dataflow-Modells, die Fähigkeiten der Hardware plattformunabhängig und effektiv einzusetzen um die Paketverarbeitung zwischen der Laufzeitumgebung und des NIC aufzuteilen. Die Laufzeitumgebung kann dann automatisch den Netzwerkstapel aufgrund dieser Informationen spezialisieren.

Des Weiteren erlaubt das Dataflow-Modell die systematische Erschließung aller möglichen Hardware-Konfigurationen und erlaubt es zu beurteilen, wie die verschiedenen Hardware-Fähigkeiten im Kontext von verschiedenen Anwendungen eingesetzt werden können.

Diese Dissertation zeigt die Effektivität des Dataflow-Modells durch die Implementierung von mehreren abstrakten Strategien zur Verwaltung von unterschiedlichen Hardware-Ressourcen mit zwei unterschiedlichen NICs.

Acknowledgments

First of all, I thank my advisor Timothy Roscoe, for giving me the opportunity to do the research, and the valuable guidance and advise. I also thank Kornilios Kurtis and Antoine Kaufmann for the collaborative work on the Dragonet project.

I thank Gustavo Alonso, Kornilios Kurtis and Andrew Moore for being part of my examination committee, and giving me valuable feedback on the thesis.

I am grateful to the whole Barrelfish team, in particular Stefan, Gerd, Simon, David, Reto and Jana for all the help throughout my PhD. In addition, I thank Claude, Besmira, Darko, Anja and Frank for their valuable feedback on the thesis. Further, I thank Simonetta for all the support in administrative issues and helping me improve the English of my thesis. I also thank Indrajit Roy, and Kimberly Keeton from HPELabs for their guidance and help during and after my internship. I extend my gratitude to everyone in the Systems group for the great time I had here.

I thank Nina for the company, and for trying to cheer me up in the difficult times. I am grateful to my family, for supporting me to reach here, and for setting me free to explore.

Finally, I thank the reader for the interest in this work.

Contents

Chapter 1

Introduction

This dissertation takes a fresh look at the design of the host network stack with the aim of effectively exploiting the increasing capabilities of Network Interface Card (NIC) hardware and systematically handling diversity and complexities associated with these hardware capabilities.

1.1 Motivation

Networks are getting faster. CPU cores are not. If computers are to handle future bandwidth and latency requirements, they will need a combination of parallelism across cores and specialized network interface hardware.

Software alone will not be sufficient to deal with the increased network speed. Parallelism can help to process more data in time, but without specialized hardware, processing of incoming network packets before demultiplexing (and of outgoing ones after multiplexing) executes serially in software, leading to basic scalability limits via Amdahl's law [Rod85]. Worse, multiplexing aside, the limits of multicore scaling [EBSA+12] will ultimately make software protocol processing a bottleneck.

Fortunately, modern network interface controllers are equipped with hardware functionalities which can help. These devices are highly complex, and feature a bewildering array of functions: multiple receive and transmit queues, TCP offload, traffic shaping, filter rules, virtualization, etc. Most of these aim at improving performance in some common enough cases: balancing packet processing load, improving QoS isolation, reducing CPU utilization, minimizing latency, etc. We will discuss these hardware features in further details in the next chapter (2.1).

Unfortunately, using these complex NIC hardware capabilities is difficult due to the lack of support from the Operating System (OS). Host network stacks in most modern operating systems are still based on the layering concepts introduced in 1980's [LJFK86] to simplify portability across different NIC hardware and protocol stack implementations. Hence, most host network stacks still have a rigid interface between the device layer and the protocol processing layer hiding most of the NIC hardware capabilities. Excepting some ad-hoc features in Windows [Mic11], an OS protocol stack is based on a simple 1980s NIC and does not cope well with today's plethora of feature sets and their programming interfaces. Indeed, Linux explicitly avoids support for all but the simplest hardware protocol offload, for reasons that make sense from the perspective of the kernel maintainers [The09], but which do not hold in a broader perspective.

The lack of systematic support from the OS for advanced NIC features has made the problem of effectively using them even harder by leaving the support for these to hardware vendors. Support for such functionality is isolated in individual device drivers, resulting in a mess of non-standard configuration tools, arbitrary resource policy hard-coded into drivers, and in some cases completely replicated network protocol stacks.

Ideally, the OS should manage these resources, allocate them appropriately among competing applications based on system-wide policy, and providing appropriate abstractions so that applications can benefit without being rewritten for each new piece of hardware.

This dissertation rethinks the OS network stack design to manage and benefit from increasingly complex and diverse NIC features.

1.2 Thesis

This dissertation argues the following thesis:

> A flexible interface and a hardware agnostic representation of packet processing capabilities are needed to effectively use the NIC hardware capabilities.
>
> A dataflow graph-based model captures sufficient information about the packet processing capabilities of hardware and the network stack state to enable automated reasoning about the NIC capabilities in a hardware-agnostic way.
>
> A dataflow model-based interface enables sharing of fine-grained information about packet processing among the network stack layers which can be used for adapting the packet processing based on the work done by other layers.
>
> Furthermore, this approach provides systematic and automated reasoning capabilities which enable policy-based NIC resource management across different NICs in a portable way.

To prove the above thesis, a prototype network stack called **Dragonet** was implemented with the following goals:

- The network stack should have a mechanism to support diversity and complexity in NIC hardware.
- The network stack should adapt to maximize the benefits of the NIC capabilities to the applications based on their requirements.
- The network stack should provide policy-based NIC resource management in a hardware-agnostic way.

1.3 Contribution

This dissertation makes the following contributions:

Modeling Packet processing as a dataflow graph: This thesis presents a dataflow model to capture the packet processing capabilities of NIC hardware and the OS network stack. This model uses two simple abstractions to represent the packet processing as a dataflow graph and an additional abstraction to capture capabilities of the NIC hardware. This thesis shows that the dataflow model captures enough information which allows us to exploit NIC hardware capabilities by enabling automated reasoning.

This thesis also explores the difficulties in exploring the NIC configuration space and proposes optimizations to reduce the search space based on the current network state by using a hardware oracle.

Policy-based resource allocation: This thesis proposes an approach to separate the resource allocation policies from the implementation by formulating the policies using a high-level abstraction of cost-function to evaluate the resource allocation plan in a hardware-agnostic way. Using the cost-function abstraction, we transform the resource allocation problem into a search problem on the NIC configuration space.

Dataflow model-based interfaces: This thesis proposes using a dataflow model-based interfaces to pass the information about packet processing capabilities between the resource management layer and the network stack. This abstract model allows the network stack to adapt to the current NIC hardware configuration without having to worry about NIC specific implementation details.

1.4 Dissertation Structure

This dissertation is structured in the following chapters:

Chapter 2 provides the background about NIC hardware and the OS network stack and explores the difficulties in using NIC hardware features. Further, it makes a case that the layered design of a host network stack with fixed interfaces is not ideal for increasing diverse and complex NIC hardware and changing application requirements.

Chapter 3 introduces the approach of using dataflow models based on simple abstractions to capture the packet processing capabilities of NIC hardware. This chapter also explores the problems in exploring the NIC configuration space and outlines a way to reduce the search space by using network state. In addition, it introduces the dataflow model-based interfaces to share the information about the protocol processing capabilities between different layers in a hardware agnostic way.

Chapter 4 discusses how information provided by the Dragonet about hardware capabilities can be used to specialize the network stack at deployment, and to adapt the protocol processing during the runtime.

Chapter 5 shows how the Dragonet approach can separate queue management policies from the implementation in a portable way across different NICs by introducing a cost-function abstraction. Furthermore, it explains the search approach for the NIC configuration space and the performance optimizations implemented. It presents two different queue management policies on two applications for two different NICs.

Chapter 6 describes a host bandwidth management use case and how Dragonet can help in using NIC hardware capabilities for rate control to meet the application's requirements.

Chapter 7 concludes the thesis and explores the potential for the future work.

1.5 Collaborative work

In addition to my advisor Prof. Dr. Timothy Roscoe, the work presented in this dissertation is done in collaboration with other members and students of the Systems group at ETH Zurich. The Dragonet network stack has been built and evaluated in collaboration with Dr. Kornilios Kourtis, Antoine Kaufmann, and Prof. Dr. Timothy Roscoe. Antoine Kaufmann lead the development of the Dragonet runtime system in his master's thesis [Kau14], and Dr. Kornilios Kourtis lead the efforts in optimizing the configuration space search [KSKR15].

1.6 Related Publications

The work presented in this thesis is part of the Barrelfish OS research project, the Dragonet network stack, and depends on and is supported by the work of others. Some of the work this thesis depends on is published, and is listed here for reference:

[KSKR15] Intelligent NIC Queue Management in the Dragonet Network Stack Kornilios Kourtis, Pravin Shinde, Antoine Kaufmann, Timothy Roscoe. In proceedings of the 3rd conference of Timely Results in Operating Systems (TRIOS-2015)

[SKKR13] Modeling NICs with Unicorn. Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, Timothy Roscoe. In proceedings of the 7th workshop on Programming Languages and Operating Systems (PLOS-2013)

[SKRK13] We need to talk about NICs. Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, Stefan Kaestle. Accepted in the 14th USENIX conference on Hot topics in operating systems (HOTOS-2013)

[Kau14] Efficiently executing the Dragonet network stack. Antoine Kaufmann. Master's thesis, ETH Zurich, September 2014

Chapter 2

Background

In this chapter, we provide background and the driving forces behind the increasing complexities in NIC hardware (2.1) as well as the current state of the host network stacks (2.2). Then we discuss the layered design of the host network stack and how this affects the efficient use of the NIC hardware (2.3). We take a use case of programmable NICs with onboard FPGA to show how network stack structure affects the potential benefits of a programmable NIC (2.4).

2.1 The state of NIC hardware

The trend of using NIC hardware to help the CPU in dealing in increasing network speeds is not new. Benefits of using NIC hardware for simple CRC calculations have been observed in past [vEBBV95] As the network speeds increased from 100Mbps to 1Gbps, hardware support for interrupt throttling [ST93] was introduced to reduce to load on the CPU. Arsenic [PF01] explored the benefits of using the NIC hardware for demultiplexing the packets.

Early NICs also reduced CPU load by offloading IP checksum calculation

and validation of packet lengths, protocol fields, etc. The value of this functionality by itself is now debatable as CPUs are much faster, but it remains as a prerequisite for further acceleration.

Starting from relatively simple TCP Segmentation Offload (TSO), vendors have implemented increasingly complex TCP Offload Engines (TOEs), experimenting with different mixes of software and hardware functionality, including the use of embedded processors [JCKL11] and intelligent coalescing of interrupts based on packet properties [WWL05]. The problem of handing off connections between OS and offload stacks has also been investigated [KR06]. TOEs can improve web server capacity by an order of magnitude [FBB+05], but they remain controversial due to complexity [The09] and limits to applicability [Mog03], an issue we return to in Section 2.2.

Multicore processors have also led to NIC features to aid scalability: multiple hardware queues for send and receive, Receive-Side Scaling (RSS), and configurable per-queue interrupt routing. The value of such features has been shown in many scenarios, for example, Routebricks provides a detailed analysis for software routers [DEA+09]. Affinity-accept [PSZM12] uses multiple queues and receive-side filters to extend the accept system call to favor the connections on local queue to improve the scalability of the accept call at the cost of minor changes to POSIX socket semantics.

Networks are now so fast that the memory system can become the bottleneck. NIC hardware is helping with this bottleneck by introducing techniques (e.g., Direct Cache Access (DCA) [HIT05] and Data Direct I/O (DDIO) [Int12]) which can deliver incoming packets to the data cache, bypassing RAM and thereby reducing memory bandwidth and latency, although performance is highly sensitive to correct configuration [KHM09].

At the same time, the CPU-intensive nature of cryptographic operations has led to some NICs integrating SSL and other crypto accelerators onto the data path [BK03]. The high performance network communication mechanisms like RDMA [RDM] and iSCSI [Net04] are designed assuming hardware acceleration, and supercomputers employ NICs with hardware support for the Message Passing Interface (MPI) [Mes09].

The rise of virtualization has also led to NIC support for virtual devices that can be directly mapped into virtual machines [PCI10, RS07]. Plentiful

hardware queues help to provide quality of service and isolation. Modern virtualization-aware NICs [Sol10b] also provide onboard IOMMUs for address translation and protection on DMA transfers, per-flow queuing, traffic shaping on transmit, and even built-in switching functionality [STJP08].

Given this complexity, it is unsurprising that some NICs have fully programmable processors onboard [Net08, Cav13]. Unfortunately, the complexity of programming these NICs have limited their use to prototyping complex offloading engines [Ang01], self-virtualizing interfaces [RS07] and applications like intrusion detection systems [BSR+06].

Also, NICs with configurable hardware such as FPGAs are gaining increasing interest in both research [Netb] and industry [Sol12]. The initial target for these NICs were the niche applications such as algorithmic trading [LGM+12], and these platforms are also used to evaluate the implications of offloading full applications like key-value stores on the onboard FPGA [CLW+13a] simple setup of UDP and the binary protocol of memcached. The implementation of scalable TCP stack on the FPGA [SAB+15] indicates the potential for supporting complex services on these NICs in a scalable way. The recent research is further exploring the benefits of using this hardware platform to implement the complex and latency sensitive services like distributed consensus [ISAV16]. The introduction of these hardware platforms into the data centers for complex and latency sensitive data center applications like web search [PCC+14] also indicates the potential benefits of capabilities of these programmable NIC platforms.

The network switch hardware is also becoming increasingly programmable to support Software Defined Networking (SDN) [MAB+08]. The usefulness of the partially programmable networking hardware is shown by the ServerSwitch [LGL+11], which uses hardware present in the network switches to implement and evaluate the data center networking designs.

In summary, hardware vendors are incorporating ever-more sophisticated capabilities into NIC hardware, and this trend is growing. Unfortunately, in most cases the benefits of these evolving hardware capabilities are not automatic, and careful system-specific configuration may be needed to fully realize them.

2.1.1 Lack of standards and abstractions

The changing hardware landscape typically needs new approaches and abstractions to effectively benefit from them. Graphics Processing Unit (GPU) hardware evolution has benefited from the SIMT (Single Instruction Multiple Threads) model and the OpenCL framework [SGS10] to simplify the use of the GPU hardware capabilities in a portable way.

Similarly, the Message Passing Interface (MPI) standard [Mes09] has helped the evolution of RDMA capable hardware, while simplifying the portable use of these hardware capabilities. The Portals network programming interface [San14] is extending these interfaces further to facilitate the future evolution of the hardware.

The recent advent of Software Defined Networking (SDN) [MAB+08], which aims to open the capabilities of network switches is benefiting from the OpenFlow standard [spe12] to facilitate the development of hardware capabilities.

Unfortunately, Ethernet NIC hardware does not have any viable standard nor the interfaces to guide the evolution of NIC hardware capabilities. This lack of standards and interfaces has lead to the increasing diversity and complexity of the NIC hardware while applications are not able to use these hardware capabilities in a portable way.

2.2 The state of OS network support

Modern OSes do not provide a good framework for NIC accelerators, and TCP Offload is a good example which illustrates the wider issues surrounding OS support for new NIC functionality. The Linux kernel network stack has limited support for TCP Offload Engines (TOEs) in favor of reducing the complexity in the network stack [The09], and hence many of these features are pushed down to the device drivers with non-standard interfaces to control them [xAp13]. Similar arguments are surveyed previously [Mog03], and we discuss few of these arguments here.

Deployment issues: These arguments can take many forms: Intelligent NICs use proprietary firmware and hence the deployment is dependent on the hardware vendor for maintaining the compatibility with the changes in the OS and fixing security bugs. Consequently, the OS is not granted the visibility into, and control over hardware, and hence OS can't manage these resources properly.

Performance gain is short term: The argument goes that cores will always get fast enough to render any hardware feature irrelevant for performance. This argument does not hold anymore in the current era of multicore CPUs where individual cores are not getting faster [Mim10]. Even the benefits from multicore CPUs are limited as the packet processing involves serial operations like demultiplexing incoming packets and any attempt to move this into software results in a serialization point where Amdahl's law fundamentally limits scalability. Moreover, with processor architecture moving towards large sets of specialized processing elements [EBSA+12], we can view NIC hardware as one subset of this trend. We need to understand how to write OS software for this kind of system.

Benefits of hardware features are unclear: Mogul [Mog03] observes The hardware interfaces were often poorly designed for performance under many common workloads. For example, even though RDMA capable hardware can be useful in some use cases, the OS overhead of descriptor management can frequently outweigh the gain from the hardware in many other cases [FA09]. As hardware features rarely improve performance in all cases and need to be used judiciously. One can even build models to predict the value of protocol offload for different applications [SC03].

The hardware lacks functionality: Modern OS protocol stacks are feature-rich, with sophisticated filtering and scheduling, and uniform configuration interfaces. In contrast, NICs which try to assume such functions may not support the full capabilities of the OS stack, can have hardware-specific resource limits (e.g., on number of flows), can suffer remote resource exhaustion attacks, and often require special tools for configuration.

Integration is difficult: Supporting a complex piece of NIC hardware in a modern OS requires large engineering efforts [The09]. In addition, having some flows or state handled by hardware and some by the OS eliminates the global system view, making it hard to manage resources.

Above reasons have lead to poor OS support for complex NIC hardware capabilities. The limited support for incorporating the complex NIC functionalities into the OS has pushed NIC vendors towards using alternate approaches. NIC vendors can *encode policy in the driver* and hide it from the OS. For example, Intel i82599 10GbE Linux driver monitors TCP connections and try to dedicate a queue to a connection by sampling the outgoing packets to determine active flows (default sampling rate: 20 packets). The OS has no control over this policy and cannot connect it with other resource allocation decisions. The other approach for NIC vendors is to expose the functionality to specially-written applications using a *non-standard control interface* [Int09] or, in some cases, a new, separate network stack [Sol10a]. This works in certain application areas (Finance, HPC), but is not a long-term solution.

There is a tread-off between hiding the hardware complexity for portability and simplicity, and exposing hardware capabilities for performance and efficiency. So far, the OS network stack design was inclined towards simplicity and portability, but as the NIC hardware is becoming increasingly capable we need to rethink

2.2.1 Motivation to rethink network stack design

There is a tread-off between hiding the hardware complexity for portability and simplicity, and exposing hardware capabilities for performance and efficiency. In section 2.2, we have seen that the OS network stack design is inclined towards simplicity and portability. But, as the NIC hardware is becoming increasingly capable we need to rethink the network stack architecture with a motivation of exploiting the hardware capabilities effectively in a hardware-agnostic way to deliver good performance to the application.

The trends in virtualization, multicore processors and NIC hardware features are showing that NIC hardware features are here to stay, and the complexity will keep increasing. The applications can benefit from these hardware features if they are properly used. For example, virtualization support on NICs [DYL+10] could deliver benefits to a single OS via reduced CPU usage and performance isolation, as with other virtualization support [BBM+12, PLZ+14, BPK+14]. The benefits for hardware queues and fil-

tering capabilities could be used for scaling packet processing with cores [PSZM12, JWJ+14, HMCR12]. Unfortunately, most of these solutions are NIC-specific, making it difficult to develop applications which can exploit NIC capabilities across different NICs. This portability problem is partially due to the OS not managing the NIC hardware capabilities, leaving it to the applications and NIC vendors to work around the OS.

2.2.2 NIC capabilities as first class OS resources

Typically, the OS would consider only few network related resources for resource management. A typical network resource includes a network endpoint address (e.g. IP address, port numbers) which are needed for sending or receiving packets. These addresses are typically managed by the network stack to control the access to packet sending and receiving capability. Also, the network bandwidth available to a host can be considered as another network resource which can be managed by the OS.

The NIC hardware is evolving to provide features like offloading, packet filtering, multiple send and receive queues, onboard cores or FPGAs, and these features can help to reduce the CPU and memory load, provide QoS and reduce the latencies. As these capabilities in the NIC can affect the application performance and overall resource utilization, the OS should be able to manage them. These hardware capabilities should be treated as first class citizens for the OS-level resource management.

We revisit the assumptions in the current network stack designs in the light of changing hardware trend of increasing NIC complexity and diversity. In next section, we discuss the motivation and reasoning for the layered design of host network stacks, and how that design is limiting the benefits of the advanced hardware capabilities in modern NICs and treating these NIC capabilities as first class resources.

2.3 Layers in traditional network stacks

The evolution of the layered network stack can be traced back to the 1980's [LJFK86] era of simple NICs and single-core CPUs. In that period, one of the main motivation for a network stack design was portability. These portability concerns lead to layering the network stack into the device-driver, the protocol processing layer, and the socket layer. These layers facilitated the portability of applications across different OSes by using the application-level POSIX interface [POS93], and the portability of the OS on different NIC hardware by using the device-level interface. Also, this layering structure mapped well on ISO/OSI standard [Zim80].

The advantages of the layered architecture of network stacks are stable interfaces and separation of concerns between different parts of the network stack. This separation and stable interfaces allow each part to evolve independently. For example, moving from 1Gbps to 10Gbps NIC hardware is possible without modifying the protocol layer or the applications. Similarly, adding new optimizations in the TCP protocol can be done without having to worry about the NIC hardware and the applications using it.

The layered architecture of the network stack has served well in shielding the applications and the OS from changes in the NIC hardware while allowing them to evolve over the time, and still remains at the core of the network stack implementations in modern operating systems.

2.3.1 Limitations of layered architecture in network stack

Part of the difficulty in exposing new hardware features to the OS comes from the lack of the flexible interface between the device driver and the OS. In addition to the inflexible interface, most protocol processing layers make implicit assumptions about separation of responsibilities between the device driver and the OS. Unfortunately, assumptions about responsibilities are so deeply embedded in the implementation of the protocol stack and the resource management policies that it is hard to adapt them for new hardware features.

For example, there are no standard interfaces to expose partial TCP offload

to the protocol processing layer and hence these features are implemented by many NICs using vendor-specific interfaces [Mog03]. But, this does not work well without modifying the protocol layer and resource management layer to incorporate the knowledge about the semantics and implications of using the hardware capabilities [Ang01].

It is not the case that the interface between the device driver and protocol processing is not evolving at all. For example, the device driver interface in the Linux network stack has evolved to support hardware features like receive side scaling (RSS), per core receive queues and per core interrupts to achieve better scalability in packet processing [HdB13]. However, these changes are happening slowly and are not fast enough to keep up with diversity and complexity of NIC hardware changes.

The diversity in semantics of the NIC hardware features is another source of complexity. Not all NICs provide the same functionality in hardware nor are the interfaces and semantics of these features similar. Also due to a lack of proper standards and global agreements it is unlikely that the device-OS interface can be easily changed to incorporate these features. The OS has an option to have separate interfaces for each NIC and hardware feature, leading to increased complexity, or ignoring the hardware features in favor of simplicity.

As a result of this diversity and complexity, Linux kernel network stack has a limited support for complex NIC features like TCP Offload Engines(TOEs) [The09], and Windows has used an approach of pushing the complexity to user space by adding TOE-specific interface [Mic11]. It is not clear if this approach is scalable with increasing hardware features, and also it is not clear if ignoring these features is advisable [Mog03].

Using NIC capabilities from applications

As the interface between applications and the network stack is fixed [POS93], it is difficult for applications to benefit from the new hardware features without proper support from the underlying OS. Applications can cheat by either bypassing the network stack altogether or by creating a non-standard communication channel with the device to use it. Both of these approaches

make applications heavily dependent on the particular hardware features at the cost of portability, and lose the benefits of using the OS for resource management. We need a flexible network stack design which can simplify utilizing the NIC hardware features, without forcing hardware-specific interfaces onto the application.

2.3.2 Background on flexible network stack design

There have been attempts to avoid the limitations of layering by re-structuring the host network stack to improve the performance. U-Net [vEBBV95] was motivated to simplify implementation of new protocols and improve the performance by limiting the kernel to perform demultiplexing and by pushing the protocol processing into the applications. Arsenic [PF01] used hardware support from NIC to expose the virtual interfaces to applications, and hence collapsing the layers in the traditional layered architecture. The exokernel [EKO95] approach is similar and it exposes the low-level NIC hardware interface directly to the applications to further explores potential for application-specific optimizations in the protocol stack.

Similarly, recent research in decoupling the control plane and the control plane at the OS level [PLZ+14, BPK+14] is revisiting layers and interfaces in the network stack in the context of increasing support for virtualization in NIC hardware. This approach separates the safety critical *control path* from the performance critical *data path* and provides a direct data path to the applications by exposing the virtualized NIC (VNIC) interface supported by the NIC hardware. The FlexNIC project [KPS+16] provides a flexible DMA interface to a network stack and applications to offload some of the application-specific packet processing onto the programmable NIC hardware with reconfigurable match table (RMT) [BGK+13] capabilities.

This related work shows the benefits of having flexibility in the network stack for exploiting the NIC hardware capabilities efficiently. Next, we present a case study of difficulties involved in using hardware capabilities in the NIC in the layered network stack architecture.

2.4 Case Study: Programmable NICs

In this section, we take a look at a NIC with an onboard FPGA as a representative of a NIC with complex hardware capabilities. We evaluate the benefits of using such hardware capabilities and the difficulties of using them.

2.4.1 Background

As networks are becoming faster, NIC vendors are exploring ways to reduce the load on the CPU by providing the possibility of full programmability on the NIC itself. As a result, nowadays we see NIC hardware with an onboard core or FPGA commercially available for high-end servers [Sol13, Int03, Net08, Cav13]. Currently, these products are targeting specialized use cases like line-rate deep packet inspection [BSR+06], and high-frequency trading [LGM+12], where the whole system and the application are custom-designed around the capabilities of networking hardware.

This trend of providing general purpose processing on NIC hardware is currently going in two directions.

The first direction is NICs with **onboard processors**, which have a small general-purpose CPU directly on the NIC hardware. This processor has direct access to the incoming and outgoing packets, and can perform inline packet processing. An example of such hardware is the Intel IXP [Int03]. This hardware has been explored for use cases like intrusion detection system [BSR+06].

The second direction is NICs with **Onboard FPGAs**. This FPGA has access to the packet flow going through the NIC. The onboard FPGA can be programmed to perform different functions based on the current requirements. Examples of this approach are the NetFPGA [Netb] project and Solarflare ApplicationOnload Engine (AOE) [Sol13] NIC.

2.4.2 Motivation for programmable NICs

Programmable NICs provide general purpose processing capabilities directly on the NIC. These processing capabilities can be used for functions like packet filtering, classification, aggregation, compression/decompression, encryption/decryption, packet modifications or application-specific processing.

Programmable NICs can be beneficial in many ways. For example, these NICs can improve the overall latencies of the key-value store by offloading the full application onto the programmable NIC [CLW+13b]. The NIC can directly generate and send a reply based on the application logic without having to communicate with CPU, and hence, can gain the advantage of reduced latency.

Onboard processing capabilities also provide a way to process data on the NIC itself. Directly processing packets on the NIC can reduce the load on the CPU by sparing it from doing this work. For high-speed data processing systems, the ability to perform extra processing on incoming data without adding significant delays or additional data movements can be an advantage. Recent research demonstrates the benefits of offloading complex services like distributed consensus [ISAV16] on network-attached programmable FPGAs. This hardware is also finding its way into data centers as accelerators for the latency-sensitive applications like a web search engine [PCC+14].

2.4.3 Evaluation

In this section, we describe our experience in evaluating a programmable NIC in accelerating the application logic. Our goal for this evaluation is to understand whether the complex NIC hardware can provide benefits to applications by using programmable NICs as a representative.

As programmable NICs can be used in many ways, we focus on utilizing it for reducing the load on the CPU by performing application-specific data processing on the NIC. This reduced load on the CPU should translate into

improvement in application performance. We measure this improvement in application performance to evaluate the benefit of the programmable NIC.

Our experiment platform: applicationOnload Engine (AOE)

In our experiments, we use Solarflare ApplicationOnload Engine (AOE) [Sol13], which provides the onboard Altera Stratix V GX A5 FPGA. This NIC has the capability of running custom application logic on the FPGA as a module which will have direct access to incoming and outgoing packets. The application module can be programmed to take actions on the packets based on their content. In addition, this platform provide services and infrastructure to facilitate using existing functionalities of the NIC. It includes the *Solarflare Board Services* which provides an interface allowing application modules to access the incoming and outgoing packets at various parts of the processing. This interface allows the application logic to use the existing packet processing functionalities of the NIC without having to re-implement them from scratch.

The AOE platform differs from the NetFPGA platform which is designed to provide community-based research infrastructure for hardware platform with programmable capabilities in form of onboard FPGA [LMW+07, ZACM14]. In addition, the NetFPGA platform provides many building blocks, which are geared towards flexibility and ease of understanding. We decided to use the AOE platform for few engineering reasons, including the availability of commodity building blocks tailored for performance, and this platform is designed for a specific configuration of offloading application logic which was suitable for us. The AOE platform also provides the infrastructure in the form of Firmware Development Kit (FDK) to streamline application development.

Even with the infrastructure support available with AOE platform, developing an application for this FPGA is a complex task and involves understanding many platform-specific components. For example, developer needs to write a Verilog code, needs to understand how to use the Qsys interfaces to communicate with other modules, understand the Solarflare Hardware Abstraction Layer (HAL) and board services to use the existing functionalities provided by the platform. In addition, the developer also need understand

AOE driver interface and how this platform communicates with the host to use it correctly.

Selecting a workload

We are targeting a typical workload of a key-value store or a business rule engine which is receiving the requests over the network and is processing them to generate the corresponding response.

In these types of systems, there is a certain amount of processing needed for each request/event which typically arrives in the form of a network packet. In the case of a key-value store like *memcached* [Dan13], this processing will involve parsing a packet to get a key and hashing it. In the case of event processing system like *Drools* [Red], this processing will involve applying certain filters on the packets to classify the event.

For our evaluation, we do not aim to remove the CPU from request/event processing by offloading the entire application onto the NIC. Instead, we focus on accelerating the application by offloading a part of the request/event processing on the NIC to reduce the load on the CPU. The reasoning behind not offloading the whole processing on NIC is that such processing can be arbitrarily complex, and typically will need access to state which is difficult to maintain on resource-constrained programmable NICs.

Another factor to be taken into account when offloading the application processing on NICs is the complexity of programming the NIC environment. Due to various resource constraints and specificity of the programmable hardware used in the NIC, programming them do perform a complex request processing or event handling involves can involve lot of engineering efforts. In addition, general-purpose CPUs can be much more efficient for complex code-paths due to their super scalar architecture, large caches and hardware optimization like branch predictions. Hence focusing on offloading only the performance-critical parts of the application logic on the programmable NICs, and using general-purpose CPUs for rest of the processing can provide better tread-off between the performance and engineering efforts.

We have used following guidelines to split the work between the CPU and a programmable NIC in our benchmarks.

- Move the parsing of a network packet on the programmable NIC to fetch only those parts which are of interest to the application.
- Move trivial processing of the incoming data to the programmable NIC, which can be done with minimal application state.

The rest of the processing is left to the CPU, which can use the partially processed data to avoid parsing the packet data again.

As an example, the memcached workload can be helped by offloading the key hashing computation to a programmable NIC. A programmable NIC can parse incoming requests, fetch the key and hash it with a minimal state from the application. This can reduce the load on the CPU running the application as it can directly use the hash calculated by the NIC.

Another example workload is rule-matching engine used by an event processing system like *Drools* [Red]. In such system, a rule-engine is used to process the content of incoming events by performing a large number of string comparisons to find the set of rules which are satisfied by the incoming event. These events typically arrive over the network, and a programmable NIC can be used to decode the events and to perform the string comparisons on these events to reduce the work of the CPU. As the set of rules are fairly static in these systems, string comparisons needed can be worked out by analyzing the set of rules. These string comparisons needed for the rules can be offloaded into the programmable NIC as the application state, and we can update these comparisons in the programmable NIC whenever the rules change. We emulate this workload in our evaluations.

Micro-benchmark details

We have designed our micro-benchmark with a goal of mimicking a typical event processing server application with request-response communication, as it provides opportunities to benefit from offloading part of the processing on NIC hardware. Our micro-benchmark can be considered as open-ended re-implementation of the event processing server as we do not fully emulate the behavior of the event processing server.

We have used a network benchmarking tool called netperf [netc] which has a support for benchmarking the request/response behavior. We have used

the UDP protocol for all of our benchmarking to avoid the complexities related to the TCP protocol. Our desired behavior is partially provided by the `UDP_RR` test of the netperf benchmark, which implements UDP based request/response micro-benchmark. It also allows configuring the request/responses sizes and sending a batch of requests before waiting without waiting for the response. The `UDP_RR` test does perform any processing on the content of the received messages in the default behavior. We modified this default behavior to perform processing on the content of the incoming messages. Our processing emulates behavior of the rule matching engines commonly present in the business rule engines. Our rule matching emulation uses specific parts of the packet to find a rule that is satisfied by the packet content by performing a string comparison operation. The number of string comparisons typically increase based on the number and complexities of these rules. We have tried to mimic this behavior by configuring netperf to perform a configured number of string comparisons on the contents of each packet.

Our goal was to saturate the server CPU, so we used a workload of 64-byte requests and 8-byte response, and configured the client to have batch of 10000 requests in flight. This load was high enough that it always kept the server CPU utilization to 100%, even though the network bandwidth used in our benchmark was not very high. We used the number of completed transactions per second by the server (shown as `Transactions/Sec (TPS)` in the graphs) as a way to measure the performance of the server, and we use the standard deviation in the TPS as error bars in our graphs presented in this section. We have measured the performance with and without FPGA processing for varied the number of string comparisons (shown as `Comparisons` in the graphs).

The FPGA application details

To show the benefits of using programmable NICs for computation, we developed an application for the onboard FPGA on a Solarflare NIC.

This FPGA application parses the incoming traffic to separate the UDP traffic coming for the specific port (used by our server), and then further processes it by extracting the key, which is then matched with pre-configured

keys to figure out which one matches. The result of the match is written into the packet at fixed offset which the server application running on the CPU can directly use, without having to do the whole key matching again.

This way, all computations related to parsing and matching the key are offloaded to the NIC. The server application running on the main CPU then just fetches the answer computed by the NIC, and then can use it to continue further processing.

Due to our design choice of partial application offload, the data path in our setup includes processing on both FPGA and the CPU for each packet. This design choice also implies that our transaction rates are much lower than a system which is fully offloaded onto the FPGA. On the other hand, our setup has advantage of ease of engineering and deployment as most of the application is developed and deployed on the CPU, and only small part of the application logic needs to be offloaded onto the FPGA.

Limitations of our FPGA offloading implementation: Due to the difficulties in programming the FPGA, the implementation we used for offloading the processing to the FPGA is quite straightforward. It can only match pre-configured keys, and it uses part of the packet to store the results of the computation instead of providing the results separately (or as part of the packet descriptor).

Evaluation: Correlation of workload and transaction rate

This experiment is designed to show the impact of increasing computational workload on server performance. We increase the amount of work our server needs to do by increasing the number of keys it should match for each incoming packet. This increased key matching simulates the workload of a business rule engine by increasing the number of business rules, where each new rule will typically lead to additional key matching for each incoming packet.

We are not using any FPGA-based offloading for this workload, and we have pinned the server application on a single CPU core. The goal here is

to see how the server application performance is impacted when the amount of work per packet increases.

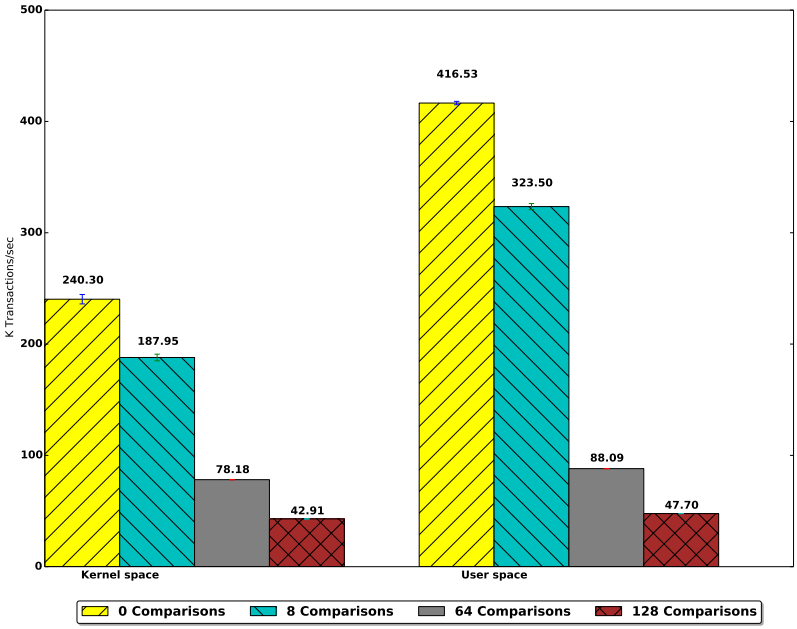


Figure 2.1: Impact of increasing CPU work on transaction rate for traditional kernel space and user space network stack

The results of this experiment are presented in Fig. 2.1. This graph shows the impact of increasing the amount of work by increasing the number of keys compared by the server for each incoming packet. We have used *Transactions per second(TPS)* as a unit of performance here.

This result shows that for a traditional kernel space network stack the transaction rate drops from 240.30K TPS to 42.91K TPS when increasing the number of key comparisons from 0 to 128 keys. This result shows that the key comparisons used in this benchmark are indeed CPU intensive, and has measurable impact on the performance of the server.

Implications of User space networking To quantify the impact of the kernel space network stack and implied context switching, we have used a user space networking implementation called onload [SC08] supported by this hardware. This user space network stack solution avoids context switching in the kernel by mapping a pair of hardware send and receive queues in the user space, and using them to directly send and receive packets.

Fig. 2.1 shows show kernel space and user space network stacks differ in their performance. The user space network stack performs better when the computational load is small (zero comparisons), and the system is processing more packets. The user space networking reaches 416K TPS whereas kernel space networking reaches only 240K TPS. The user space network stack benefits by avoiding system calls related to sending and receiving packets, and hence can do more relevant work leading to higher transaction rate.

When the number of key comparisons is high, and the CPU is responsible for this work, the advantage of user space networking is not significant as most of the CPU time is going into key comparisons.

Evaluation: Performance improvements with FPGA

In this benchmark, we quantify the benefits which can be achieved by offloading compute-intensive work to the programmable NIC. We compare the performance of offloading the work to the FPGA (marked as `Using FPGA`) with letting the CPU do the work (marked as `Without FPGA`). We have picked the workload of 64 string comparisons for this experiment as it showed a significant impact on transaction rate in the previous experiment (Fig. 2.1).

Fig. 2.2 presents the results of this experiment. It shows how a kernel space and a user space network stack perform with and without using the NIC offloading facilities.

The results show that whenever the FPGA is used for performing the string comparisons, the performance of the server improves, compared with doing this work in the CPU. For example, the performance of a kernel space

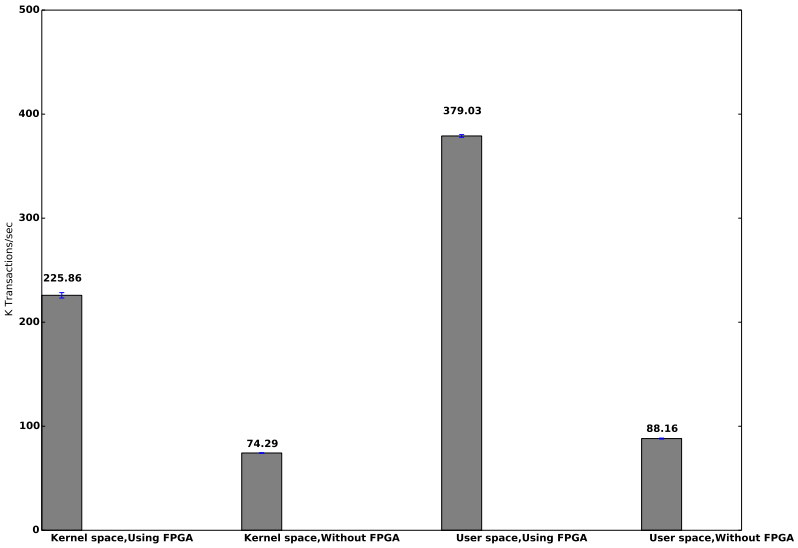


Figure 2.2: Performance improvement with FPGA for 64 string comparisons

network stack improves from 74K transactions/sec to 255K transactions/sec when using the FPGA.

The benefits are even higher for the user space network stack when using FPGAs and hence the transaction rate increases from 88K to 379K. User space networking can provide additional benefits by avoiding the system call overhead in sending and receiving the data over network. The kernel space network stack reaches a transaction rate of 225K when using FPGAs for 64 key comparisons, whereas the user space network stack outperforms it by reaching the transaction rate of 379K transactions/sec.

The performance of the server for 64 key comparisons with using the FPGA (225K TPS) is similar to the performance of the server for zero key comparisons (240K TPS) without FPGA from the previous graph (Fig. 2.1). This observation indicates that the system's performance is bound by the number of packets it can receive and send with the NIC and not on key comparisons.

Evaluation: Impact of having the FPGA on the data path

Here, we present an experiment to quantify a negative impact of having an additional processing unit on the NIC. We measure this negative impact by letting the FPGA implementation on the NIC perform the work of key-comparisons, but the server implementation ignores the work and does the comparisons on the main CPU anyway. This setup leads to duplicated work, once on the NIC FPGA and once on the server CPU. This configuration allows us to measure the degradation of performance caused by the duplication of work, and we present this additional work as the worst-case overhead of having an active NIC FPGA in the packet processing path.

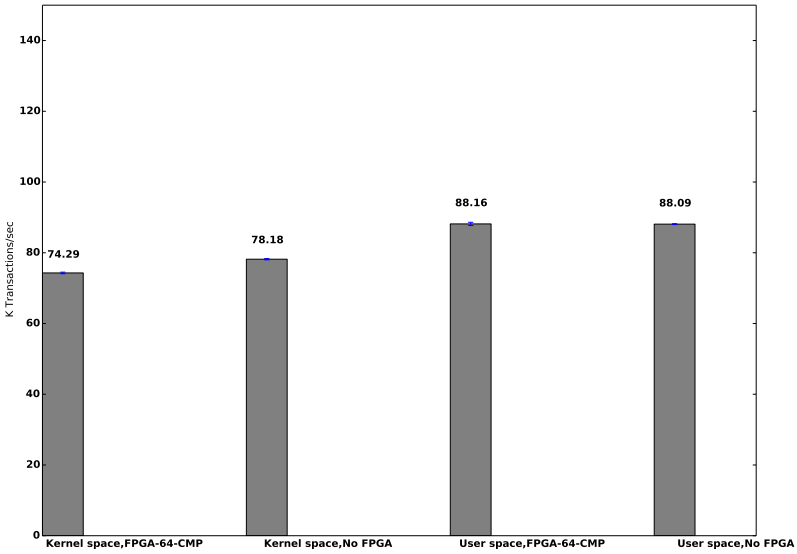


Figure 2.3: Impact of FPGA processing (`FPGA-64-CMP`) and no FPGA processing (`No FPGA`) on application performance

Figure 2.3 shows the results of this experiment. In the graph, `No FPGA` represents a run where the NIC FPGA is not doing any work on the packets, and `FPGA-64-CMP` represents the configuration where the driver is performing key-comparisons which are then repeated by the CPU. This configuration makes sure that the CPU is doing the same amount of work in

both cases, and `FPGA-64-CMP` configuration is doing additional work in the NIC FPGA. The results show that the transaction rate reported in these two settings is similar for both the kernel space network stack and the user space network stack. This result indicates that doing extra work of key comparisons in the NIC FPGA does not have any observable effect on the transaction rate reached by the server.

2.4.4 Observations about using programmable NIC

These experiments show that offloading part of the application processing on a programmable NIC can lead to performance improvements by freeing the CPU for doing more application-specific or other work.

These benefits come with some limitations. Currently, the application has to be written for the specific capabilities of a particular programmable NIC. This approach pushes all the hardware related complexities in the application development, and makes the application non-portable. Also, application developers need to think about which processing should be offloaded to the NIC, and when.

The interfaces used for communication between the application and the NIC are vendor-specific and typically OS will not be aware of them. Such use of vendor-specific interfaces implies that the OS will not be able to provide any resource management, pushing the responsibility of sharing the resources to the application as well. A typical deployment will involve a single application owning the NIC hardware and bypassing the OS completely to utilize such a programmable NIC. Vendors tend to provide their own mechanisms to share the NIC hardware between multiple applications, or such sharing is not supported at all.

Most of the problems listed above stem from the inflexible interface between the NIC driver and the OS, and lack of mechanisms to inform the OS about hardware capabilities in a more systematic way. Hardware vendors are forced to bypass the OS to provide their services to applications using their custom tools, and in process locking the applications into their proprietary infrastructure.

2.5 Discussion

In this chapter, we made a case for how layered structure and fixed interfaces limit the ability of a network stack to adapt based on changing application requirements and changing hardware. We have presented a case study using programmable NICs to show the potential advantages and difficulties in using programmable NIC hardware.

A layered architecture and fixed interfaces used in a typical network stack implementation is designed for the portability, and it limits the visibility of the full functionalities in the other layers. This limited visibility in the capabilities of other layers has an implication in using these capabilities effectively. In the case of the host network stack, the increasingly diverse capabilities in the NIC hardware are behind the device-driver interface, and due to the limited visibility, a host network stack and the OS can not reason about these capabilities, or manage them efficiently.

We believe that a layered network stack structure with fixed interfaces is not suitable to adapt with hardware capabilities and application requirements. We need to provide an adaptable division of responsibilities and a flexible interface to allow such adaptation based current hardware capabilities and application requirements.

We need a hardware-agnostic way to capture and share enough information about the NIC hardware capabilities and the current network state. With access to information about capabilities of the hardware, we can develop a network stack which can adapt itself to exploit hardware capabilities based on application requirements.

In the remainder of the thesis, we present a network stack aimed to tackle these challenges. We present the Dragonet network stack which provides a way to capture the capabilities of the NIC hardware and the network state, and a way to analyze them to provide a networking solution optimized for current application requirements.

Chapter 3

Dragonet

3.1 Introduction

Modern host network stacks are evolved from an era when NICs were simple and computers had a single CPU. As a result, their primary design goal was to provide an efficient and portable software implementation of network protocols across different NIC hardware. This lead to the separation of the hardware dependent components in the device driver, and the use of simple and minimal interface to send and receive packets from the host network stack. However, changes in hardware trends force us to rethink the design of the host network stack.

Today's hardware is fundamentally different in two significant ways. First, it exposes a high degree of *parallelism* to the software. Multi- and many-core machines are an important aspect of this trend, yet not the only one. Modern NICs offer a large number of hardware queues and filters which can be used to distribute network processing on multiple cores to achieve better performance.

Second, in an attempt to address power efficiency challenges, current hardware is increasingly *specialized*. Specifically in networking this manifests

as the rich and diverse features of protocol offload functions provided by modern NICs, ranging from checksum and segmentation offload to full protocol offload.

Since the individual CPU speed is not increasing [Mim10], exploiting specialized hardware might be the only way to support increasing network speeds. Hardware offload functions, however, are not always beneficial [Mog03]. Hence, the network stack should not only allow the use of NIC hardware features, but also allow for reasoning about their benefits.

Overall, in addition to implementing network protocols, we identify NIC resource management as a second major concern for network stacks. In coordination with the rest of the OS, a network stack needs to reason about, efficiently manage, and correctly use hardware resources such as NIC queues, offload functions, and cores.

Unfortunately, efficiently using the NIC hardware features is not trivial due to the complexity of the hardware and their interactions with other hardware features. The network stack needs to have knowledge of the semantics of these complex hardware features and the implications of using them. Also, the diversity in NIC hardware features increases the difficulty of using these capabilities.

In this chapter, we present **Dragonet**, a new approach to building a host network stack that attempts to address these concerns. The objective of the Dragonet approach is to enable the host network stack to exploit the NIC hardware capabilities in a hardware-agnostic way. We currently target Quality of Service (QoS) and bandwidth controlling hardware capabilities, and this approach can further extended to improve latencies as well. We aim to provide management of network resources based on high-level policies by systematically searching the NIC configuration space.

Dragonet introduces two changes to implementing a network stack. First, it uses dataflow models to capture the capability of the NIC hardware and the state of a network stack. These models enable automatic reasoning for configuring the NIC hardware and customizing the network stack execution to achieve specific user requirements.

Second, Dragonet offers new interfaces based on the dataflow model which allow adaptable network stacks to easily exploit NIC hardware in a portable

way.

In this chapter:

- We propose a dataflow-based model (3.3) that provides a unified way to abstract NIC hardware and network protocols, aiming at addressing the increasing complexity and diversity of NIC hardware.
- We explain the problem of exploring the NIC configuration space (3.4) and propose using a hardware oracle-based approach (3.4.1) to simplify the problem by reducing the configuration space.
- We propose an interface based on cost-function (3.6.1) as a mechanism for policy-based NIC hardware resource allocation (3.6).
- We propose a new interface between the NIC device driver and the network stack based on the dataflow model to simplify using hardware capabilities in a portable way (3.7.1).

3.2 Background

The techniques we use in this work are build build on existing areas of related work. We discuss these ideas of modeling packet processing in network stacks organized as graphs and declarative techniques for dealing with hardware complexity.

3.2.1 Modeling packet processing as graphs

Modeling is a useful technique to analyze certain properties or to provide reasoning about a software or hardware system. The abstractions used for modeling dictates what type of reasoning it can support. For example, the Kahn Process Model (KPN) [Gil74] can capture and reason about parallel computation performed by autonomous systems communicating with each other. Synchronous languages are used to model and verify the reactive systems which are continuously reacting with environment [Hal98]. These models are focusing on capturing concurrency and providing reasoning about the correctness and they are not well suited for modeling the

packet processing which typically has little concurrency and hence need different approach.

The x-Kernel [HP91] uses the approach of modeling the protocol processing in the network stack as a graph in order to simplify the development and composition of new network protocols by using a common set of abstractions. Each protocol (e.g., IP, UDP, TCP) in the x-Kernel represents an object, and these protocol objects are organized at kernel configuration time to compose the network stack.

The Click soft router [KMC+00] uses a model based on directed-graph to build flexible and configurable software routers by assembling simple packet processing modules. These modules are called *elements*, and each element implements a simple packet processing function as a `C++` object.

The SoftNIC approach [HJP+15] uses a dataflow graph model to create a modular packet processing pipeline. Each module in this pipeline implements a NIC feature. The pipeline is assembled based on the features which are available in the NIC. The SoftNIC utilizes the Intel DataPlane Development Kit (DPDK) [int] to query and program the NIC features in the hardware.

These approaches use the graph model to specify the required packet processing and to facilitate the implementation these desired functionalities. In the Dragonet approach, we use the dataflow graph model to enable reasoning about the hardware capabilities and software requirements, as well as to implement a specialized network stack.

3.2.2 DSL to capture hardware capabilities

There is a long tradition of using Domain Specific Languages (DSL) for hardware description in OS development. *Device trees* describing platform configuration are widely used in modern operating systems like Linux. The Device trees are used particularly for system-on-chip hardware [LB08]. Research DSLs like Devil [MRC+00] and Mackerel [Ros13] help simplify accessing the device registers, and Termite [RCK+09] aims to use the device specification to generate correct-by-construction device driver code. The Dragonet approach for hardware modeling is complementary: it does not

provide a hardware access mechanism, but instead provides a semantic description of the available functionality.

The P4 [BDG+14] language uses a DSL approach to model the packet processing for programmable network switches in protocol-oblivious and hardware independent way. This approach focuses on modeling protocol-oblivious packet processing using *match+action table* abstraction, these *match+action* tables are then programmed into a *Reconfigurable Match Table* (RMT) pipeline provided by the recent programmable network switches [BGK+13, Ozd12]. Our approach differs from P4 by focusing on modeling the capabilities of the NIC hardware, and then working out how to use these capabilities for the current network state. The Dragonet approach uses a dataflow graph model suited for the NIC hardware and differs from the *match+action table* model used by P4.

The FlexNIC project [KPS+16] uses an interface inspired from the P4 language to provide flexible DMA capabilities and offloading application-specific operations onto programmable NIC hardware with RMT capabilities. Our Dragonet approach differs from the FlexNIC approach as we are targeting fixed function capabilities in the NIC hardware, whereas FlexNIC aims to exploit RMT hardware support.

In Dragonet, we are using dataflow-based interfaces to pass information about current capabilities between different layers and efficiently exploit the diverse set of NIC hardware features in a hardware-agnostic way.

3.3 Dataflow Modeling

The primary challenge motivating Dragonet is NIC diversity. We tackle this challenge by making an assumption about NICs to simplify our model: we assume that it is possible to represent their functionality as a dataflow graph. Based on this assumption, we create a dataflow-based model capable of capturing the NIC functionalities, as well as the state of the network stack.

We build our models using a small set of abstractions as basic building blocks. The core of Dragonet consists of three node types: (i) function nodes (f-nodes), (ii) logical operator nodes (o-nodes) and (iii) configuration

nodes (c-nodes) . The f-nodes combined with the logical operators model the execution flow and dependencies of protocol processing, while the c-nodes capture the configuration options of the NIC.

These basic building blocks are not sufficient to fully model a NIC: for example, reasoning about performance requires more information than the PRG provides. To avoid making additional assumptions about NICs, we handle these cases by annotating nodes with attributes (3.3.5). Although we expect to build common abstractions for many of these attributes, they are external to the basic node model.

Our model is not intended to be a precise representation with all the procedural information at instruction level about the computations. Instead, it is high-level declarative description of the protocol processing with information about their order and dependencies.

We model both the network stack and the NIC hardware as dataflow graphs. We call them the *Logical Protocol Graph* (LPG), and the *Physical Resource Graph* (PRG), respectively.

The main goal for our model is to capture protocol processing capabilities and network state (discussed in section 3.3.1). In addition, our model also need to support a way reason about the network state in context of NIC hardware capabilities. We provide this reasoning by *embedding the PRG in the LPG*. The embedding algorithm maps parts of the LPG into the PRG based on the hardware capability. The mapped nodes represent protocol processing operations that are offloaded to the hardware. The remaining nodes are parts of the protocol which will not be implemented in hardware and thus must be emulated in software. We discuss embedding in more details later (3.7.2).

Our approach of using Directed Acyclic Graph (DAG) based dataflow model to represent packet processing has few implications. One of the important implication is the difficulty in representing protocol processing involving cycles. Our model does not prohibit cycles, but it does not yet provide first-class mechanisms to capture cycles and reason about it.

Next, we discuss how we model the network state using the LPG, and the NIC capabilities using the PRG. We also describe *Unicorn* (3.3.6), the Domain Specific Language (DSL) we created to simplify the development of

these models and show an example of modeling the Intel i82599 using our DSL (3.3.7).

3.3.1 Network state

Typically the host network stack needs to maintain a view of the global network state (e.g., routing table, ARP cache) and information about the local networking resources (e.g., available NICs, their MAC and IP addresses), so that the network stack knows how to communicate with other hosts. Most of this information is fairly static and is applicable to the whole machine and not just a single application.

In addition to the network and machine specific information, the network stack also needs to maintain application-specific state information about currently active flows. The application-specific state can include information about the application buffers, the CPU cores used by the application, currently active send and receive requests, flow manipulation requests (e.g., socket, connect, listen, close), application endpoint descriptors (e.g., file descriptors associated with network sockets), etc.

The network stack also need to maintain the flow-specific state including the remote and local endpoint addresses, connection type, a current state of the flow, attributes associated with the flow (e.g., bandwidth required and allocated, other QoS attributes).

Furthermore, the network stack need to maintain the protocol-specific state for each protocol used by the active flows. For example, for every TCP flow, the network stack need to maintain the TCP state information (e.g., sequence and acknowledgment numbers, buffer windows, congestion windows, timers, etc.).

We need to be able to capture this state in our model either explicitly or implicitly. We describe our approach to model the network state next.

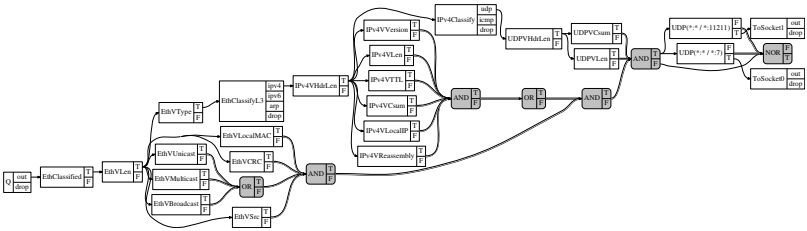


Figure 3.1: Simplified LPG example for receive side with two UDP applications listening on ports 7 and 11211

3.3.2 Logical Protocol Graph (LPG)

The LPG represents both the static parts of protocol processing (e.g., the calculation of the checksum of incoming packets), as well as the dynamic network state (e.g., the network flows as they come and go). It is a logical description in the sense that it is concerned with what needs to be executed for each packet which might be sent or received by the network stack rather than how. The LPG can be derived from the current OS protocol stack state.

The LPG is directed and, broadly speaking, represents the processing required for a packet from the NIC to reach the matching application and vice versa. Viewed from the opposite direction (i.e., by reversing the edges) the graph represents the processing dependencies of the network stack. For example, before a received packet reaches the user application, its checksum must be computed and validated.

Fig. 3.1 shows an example of a simplified LPG for a receive side with support for Ethernet, IP, and UDP processing. In this particular example, two applications, `echoServer` and `memcached`, are waiting for packets on UDP ports `7` and `11211` (depicted by nodes `ToSocket0` and `ToSocket1`) respectively.

Next, we discuss the main building blocks used in modeling the network state using the LPG.

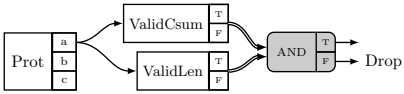


Figure 3.2: Example of f-nodes (with white background) and o-node (with gray background) as a building block for creating the LPG

Function nodes (f-nodes)

The f-nodes are the basic components of our graphs. They represent individual packet computations and are labeled based on the computation they implement. Each f-node has a single input edge and multiple output edges grouped in *ports*. An internal implementation function is specified for each f-node. Conceptually, the implementation function performs the processing on the input data and enables a single output port of the f-node. Enabling an output port results in enabling the nodes that are connected to by the edges of that port. Enabled nodes can be executed in subsequent steps. Typically, function nodes are used for implementing protocol processing, and can also manage the protocol-specific state. In the example of Fig. 3.2, the f-node labeled `Prot` checks the protocol type of the incoming packet. If the protocol is "`a`", then it enables two f-nodes which checks whether the packet has a valid length and a valid checksum (`ValidLen` and `ValidCsum` respectively).

Operator nodes (o-nodes)

F-nodes have a single input, and hence can capture only linear paths. To handle concurrent paths, where enabling an f-node might depend on the output of multiple other f-nodes, we use *logical operator* nodes (o-node).

Each o-node corresponds to a logical operator (e.g., `OR`, `AND`), and is used to combine the outputs of multiple nodes. Each of these input nodes connects two output ports to the o-node: one corresponding to *true* and one to *false*. As can be seen in Fig. 3.2, we represent these two edges with double lines originating from the `T`/`F` ports. O-nodes activate one of their output ports (`T` for *true* and `F` for *false*) based on the semantics of the logical operators

(e.g., OR, AND). In the example of Fig. 3.2, the AND's T port is enabled if both inputs are *true*, otherwise the F port is enabled. The logical operators can also be short-circuited to enable the output as soon as it has sufficient information available. In addition, if the o-node has only one input, then it is assumed to forward the input directly to the output by default.

3.3.3 NIC capabilities

We take the Intel i82599 [Int10b] as an example NIC to describe complex features and capabilities available in recent server NICs.

The Intel i82599 NIC has protocol-specific features including TCP segmentation offload, checksum calculation, packet reassembly, jumbo frames, header splitting on received packets, replicating broadcast/multicast packets, and bandwidth shaping features including transmit rate schedulers. The NIC can also do packet validations including detecting under/over size packets and CRC validations.

Furthermore, the NIC has 128 send and receive queues and different types of filters to steer the flows into these queues. These filters include 128 MAC address filters, VLAN filters, 128 5-tuple filters, a syn filter and large number of flow director filters (32K for hash filtering mode, or 8K for perfect match mode).

The processing applied on the packet and the final destination of the packet depends on both the configuration of the NIC and the packet content. We need a way to capture the protocol processing and flow steering behavior of the NIC for a given configuration. In addition, we also need a way to capture the impact of different configurations on the behavior of the NIC. Next, we describe our approach to model the NIC capabilities.

3.3.4 Physical Resource Graph (PRG)

We need a way to capture the capabilities and the current configuration of the NIC hardware so that we can reason about it. For this purpose we use

a Physical Resource Graph (PRG). Our primary goal is to facilitate the automated reasoning about hardware capabilities and the network stack state, we therefore use the same dataflow model to capture both.

There are two main aspects of the NIC hardware capabilities we want to reason about. The first aspect is the entire space of potential functionalities that a given NIC can provide. This information is needed for the NIC resource management and is useful to decide which hardware functionalities should be enabled based on the current network stack state, and to understand their implications. We use the **unconfigured PRG (U-PRG)** to capture this information.

The second aspect is the information about the specific NIC hardware functionalities which are currently configured in the NIC hardware. This information is needed by the network stack control path to adapt the data path and the packet processing in order to exploit the work done by NIC hardware. We use the *configured PRG (C-PRG)* to capture and pass the information to the host network stack (further details will be provided in section 3.7).

A simplified example of PRG for the receive side of the Intel i82599 NIC is shown in Fig. 3.5.

The PRG model is not intended to precisely represent the NIC hardware design, but rather to model the protocol processing capabilities of the NIC along with its configuration space. The PRG can be built by studying the device datasheet and the device driver. Developing the PRG is a one-time effort and can be developed with a device driver ideally by the hardware vendor.

Next, we discuss additional building blocks needed for PRG modeling, and explain how we generate C-PRG from U-PRG.

Configuration nodes (c-nodes)

Modern NICs offer rich configuration options that can drastically vary the NIC's behavior. We represent a NIC configuration and how it affects the resulting PRG using *c-nodes*.

At a high level, a c-node represents the configuration space of the NIC. Each c-node corresponds to a configuration variable. It specifies the set of possible values, and how applying a value affects the graph. Applying configuration value to a c-node results in a set of new nodes and edges that replace the c-node in the graph. Each c-node defines how the graph is modified by adding new nodes or edges, based on a configuration value. If we allow each c-node to modify the PRG in arbitrary ways (i.e., add edges and nodes everywhere in the graph), reasoning about configuration becomes challenging, especially when multiple c-nodes exist in the graph. To avoid this, we constrain c-node modifications to be local in the area defined by the c-node. Specifically, all new edges added by a c-node must have a source node which is either a new node or a node x for which an edge exists from x to the c-node in the unconfigured graph. Analogously, all new edges must have a destination node which is either a new node or a node x for which an edge exists from the c-node to x in the unconfigured graph. Under this restriction, the changes that a configuration node can apply to the graph are restrained to the nodes that it is connected with.

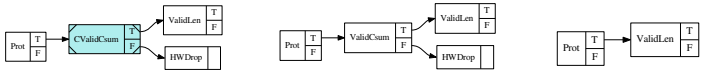
More formally, assuming a graph G (typically the PRG) with vertices $G.v$ and edges $G.e$, a *generic c-node* $x \in G.v$

consists of a configuration space C_x and a function f_x that maps each point in the configuration space $c \in C_x$ to a subgraph Γ.

Subgraph Γ consists of vertices $\Gamma.v$ and edges $\Gamma.e$, and is subject to a number of constraint: First, vertices in $\Gamma.v$ should not already exist in G ($\Gamma.v \cap G.v = \emptyset$). Second constraint, if I_x are the nodes that point to x ($I_x = \{i \in G.v \mid (i, x) \in G.e\}$)[1] and O_x are the nodes pointed by x ($O_x = \{o \in G.v \mid (x, o) \in G.e\}$), then each edge in $\Gamma.e$ should start from a node in either I_x or $\Gamma.v$ and point to a node in either O_x or $\Gamma.v$ ($\forall (j, k) \in \Gamma.e : (j \in I_x \cup \Gamma.v) \wedge (k \in O_x \cup \Gamma.v)$). When applying a configuration c to a c-node x in a graph G, G changes in two ways: (i) x is removed and vertices $\Gamma.v$ are added to to $G.v$ (ii) x's edges are removed and edges $\Gamma.e$ are added to $G.e$.

In practice, we model most of our configuration nodes using *simple c-nodes*, i.e., nodes that select one of their output ports based on the configura-

[1] for simplicity, we ignore that in actuality our edges originate from ports rather than vertices.



(a) Unconfigured c-node (b) c-node enabled (c) c-node disabled

Figure 3.3: Example of a c-node with a boolean configuration value, and a resulting C-PRG when the configuration node is enabled and disabled with a configuration value `true` and `false` respectively.

tion value. Fig. 3.3a shows an example of a simple configuration node `CValidCsum`, which can enable or disable functionality of `ValidCsum` based on the boolean configuration it provides. If the configuration is enabled, then the node will get replaced with an f-node providing the `CValidCsum` (Fig. 3.3b), and if the configuration is disabled, the configuration node will be removed (Fig. 3.3c).

C-nodes aim to address the diversity of modern and future NICs and are intended for algorithms that explore and evaluate different configuration options. Each c-node models the configuration space it supports, and the implications of applying a particular configuration to the PRG. This functionality allows a systematic exploration of the whole configuration space.

C-nodes with a small configuration space (e.g., a single register that enables or disables a NIC feature) can be naively handled by these algorithms using exhaustive search. Exhaustive search, however, is highly inefficient for c-nodes with a very extensive configuration space (e.g., a configuration for mapping network flows to hardware queues). These cases typically require additional a priori knowledge to reduce the search space. When steering network flows to hardware queues, for example, space can be reduced by only considering filters that match network flows corresponding to active connections.

An alternative approach to c-nodes could be to enforce a common abstraction on the PRG level that simplifies the search in the configuration space. However, the diversity of NICs makes it difficult, if not impossible, to devise a common and future-proof abstraction for the configuration space without substantially sacrificing flexibility. Here, we decided to take a most flexible

approach of exposing the full NIC configuration space on the PRG. Exposing the entire configuration space allows implementing NIC-specific policies that can exploit any capabilities of a NIC. At the same time, it does not exclude the possibility of building common abstractions on top of the low-level PRG interface, in a similar manner as common NIC drivers implement an OS-specific interface.

Configuring the PRG

A PRG node which has unconfigured c-nodes is called **unconfigured PRG (U-PRG)**. The configuration of the PRG is not a single step process, and each c-node can be configured separately, giving us partially configured PRGs during the intermediate steps. This process of configuring the PRG can also be viewed as progressively making the PRG concrete.

Once all the c-nodes in the PRG are configured, we get a fully **configured PRG (C-PRG)**. The fully configured PRG can be passed to the network stack to inform it about the exact processing which will take place.

3.3.5 Attributes

Our model provides the flexibility to capture additional information by annotating nodes with attributes. We use attributes as a way to extend our model with additional functionalities.

These attributes can be useful when reasoning about the model, or to simplify its execution. For example, a fine-grained performance model for the whole network stack can be built by annotating each node in a graph with their individual performance characteristics, such as CPU cycles used, throughput, and latencies.

As Dragonet matures, we expect to define common abstractions for many of these attributes. In the following paragraph, we discuss a potential use case.

Protection attributes: The execution of certain nodes in a graph is critical for the overall correctness and protection of the network stack. For example,

classifying incoming packets correctly and copying them into the application address space is a security-critical operation, and therefore needs to be executed by a codebase which is trusted by the system to behave correctly. On the other hand, calculating the UDP checksum does not need to be trusted by the system as it can only affect the correctness for a specific flow.

By annotating the nodes with attributes with their security properties, we can extend the embedding process to adapt based on the security model constraints, or the NIC hardware capabilities for protected DMA to the application address space [Sol10b]. For example, if the deployment system fully trust the application code to work correctly, then we can use this trust to allow the application to directly access the NIC hardware queues without worrying about protection related consequences.

3.3.6 Unicorn

Unicorn is a domain specific language to express the LPG and PRG so that they can be easily generated from concise descriptions, and effectively manipulated by appropriate algorithms that implement the embedding and possibly other operations.

Unicorn language has two different aspects: (i) a *concrete syntax* for writing descriptions of NICs, and (ii) an *abstract model* for representing Dragonet dataflow graphs, (PRGs and LPGs). The concrete syntax is intended for only writing PRGs based on descriptions found in the NIC vendor documentation, though we also use it to write test-case LPGs for development and debug purposes. At runtime, the Dragonet protocol stack itself will maintain the LPG as connections come and go. Since there is a close correspondence between the concrete syntax and the abstract model, we will mostly conflate them in this chapter.

Unicorn implementation

We implemented Unicorn using Haskell's QuasiQuotes as an embedded DSL in Haskell [Mai07]. Our DSL provides a convenient way to try out

LPG models for protocol processing, PRG models for NICs and embedding algorithms for network stack specialization.

We present an example showing a code snippet written in our DSL in Fig. 3.4. Most model objects are defined using a keyword followed by an identifier and a body in brackets. Keywords have the expected semantics: `node` defines f-nodes, `config` defines c-nodes, and `graph` defines graphs. The connectivity of the nodes is modeled using ports that connect to a list of nodes. For example, port `ipv4` of an f-node `EthClassifyL3_` connects to two nodes: `IsIPv4` and `IPv4Csum_`. Attributes are defined using the `attr` keyword. An explanation of `IsIPv4`'s software attribute is given in §3.3.7.

A c-node is defined using a set of ports and a configuration `type` specifying their configuration space. The `type` provides information about the configuration space of the c-node. For example, the `EthCVCRC` c-node has a simple boolean configuration space whereas `C5TupleFilter` supports configuring 128 separate five-tuple filters, each containing source, destination IP address, source and destination port numbers, protocol type, priority, and the target queue-id. This structured information about the configuration space provides a way to enumerate the configuration space.

A c-node can be configured by binding it to a specific configuration value from the supported configuration space. For example, `EthCVCRC` can be configured with a value `true` to enable the CRC checksum validation functionality in hardware, or it can be configured with a value `false` to disable this functionality. Generic c-nodes are implemented using an additional Haskell function that implements the functionality of f_x as described in section 3.3.4. For `EthCVCRC` node, this function is `configValidCRC`, and it can either replace the c-node with the f-node, or remove the c-node from the PRG based on the configuration value provided. Fig. 3.3 shows an example of how this function will work based on the configuration.

The constructs discussed above constitute the basic core of the language. We also have shorthand notations to avoid boilerplate code, to support more intuitive error messages, and to simplify the programming. For example, our prototype supports boolean nodes, defined using the keyword `boolean` instead of the keyword `node`. The boolean nodes are constrained to a specific structure: they are expected to have exactly two output ports: `true` and `false`. Another example is the `cluster` construct, which supports

hierarchical definitions to group the nodes together and avoids repeating the same prefix in node names.

3.3.7 Example: Modeling the i82599 with Unicorn

Here we discuss how we model the Intel i82599 NIC [Int10b]. Part of the Intel i82599's receive path PRG is shown in Fig. 3.5. We focus on two interesting features of the Intel i82599 from a modeling perspective: hardware checksum calculation and hardware queues. In Fig. 3.5 we present a simplified example of the U-PRG and two different C-PRGs generated using two different configurations.

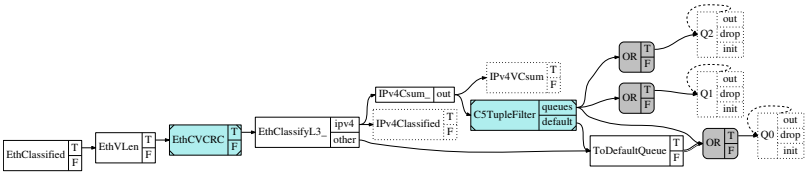
Hardware checksum calculation

The Intel i82599 NIC supports hardware checksum calculation for a few protocols (e.g., Ethernet, IPv4 and TCP), however, modeling these functionalities is not trivial. In addition to handling the configuration space with a c-node, our model also needs to handle the partial implementations and non-standard ways to access the results of the hardware computation. For instance, the NIC supports classifying IPv4 packets on the receive side and verifying the IPv4 checksum. However, the results of these computations are stored in the descriptor passed to the network stack, thus making a minimal amount of software processing necessary.

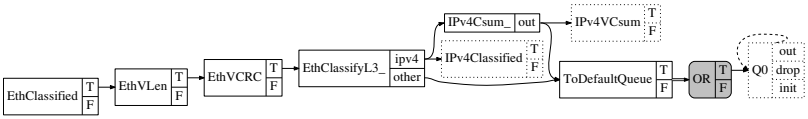
To deal with these cases, we add software nodes to the PRG. Software nodes allow expressing dependencies between PRG nodes and software nodes, thus capturing any additional device-specific software functionality required from the driver. These NIC-specific software nodes implement the required functionalities by interpreting the partial work done by the NIC hardware and implementing the rest in software. In Fig. 3.5 for the Intel i82599, the `IPv4Classified` and `IPv4VCsum` are the software nodes (in our graphs they are denoted with dashed boxes). These nodes test if the packet is of type IPv4, and the validity of a checksum by interpreting the flags set in the packet descriptor, instead of performing the complete checks on the packet itself.

```
graph e10kPRG {
     cluster Rx {

        config EthCVCRC {
                type { (enabled: Bool) }
                function configValidCRC
                port true[EthClassifyL3_]
                port false[] }

    node EthClassifyL3_ {
        port ipv4[IsIPv4 IPv4Csum_]
        port other[ToDefaultQueue] }

    boolean IsIPv4 {
        implementation E10kL3IPv4Classified
        attr "software"
        port true[]
        port false[] }

        config C5TupleFilter {
            type { (sip:   <UInt 32>,
                    dip:   <UInt 32>,
                    proto: <Enum (TCP,UDP,SCP,OTHER)>,
                    sport: <UInt 16>,
                    dport: <UInt 16>,
                    prio:  Int 1 7,
                    queue: <UInt 7>) } <,128>
            function config5tuple
            port queues[Q0Valid Q1Valid Q2Valid]
            port default[ToDefaultQueue] }

    } // end cluster: Rx
} // end graph: e10kPRG
```

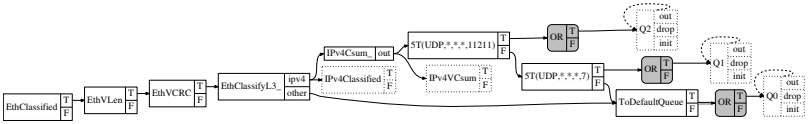
Figure 3.4: Unicorn snippet for PRG shown in Fig. 3.5a.



(a) Unconfigured PRG



(b) configured PRG with Ethernet checksum validation and no filters



(c) configured PRG with Ethernet checksum validation and two 5-tuple filters configured

Figure 3.5: Simplified PRG example of the receive side of the Intel i82599 10GbE adapter. It presents an unconfigured PRG and two differently configured PRGs based on two separate configuration values.

Hardware queues and filters

Hardware queues are a common feature of modern NICs. They provide multiple receive and transmit descriptor queues to the OS, as well as packet filtering facilities. These hardware features are mainly used for steering the flows into different hardware queues which can then be processed by separate cores, leading to better utilization of the multicore CPUs. These hardware features are becoming increasingly important for scalable packet processing and quality of service for network flows on the modern multicore architectures. The hardware queues and filters are also instrumental for recent research in the separation of data and control flow in network processing [PLZ+14, BPK+14].

Due to the advantages of hardware queues and filters, the ability to manage them is becoming increasingly important. Unfortunately, diversity in the capabilities and semantics of these filters make it difficult to manage them in a portable way. In chapter 5, we will examine the diversity of these filters (5.2.2) and the complexity involved in modeling and configuring them (5.3.1) in the context policy-based queue management. Here, we focus on showing how we can use Unicorn to model the complexities of these filters.

Modeling 5-tuple filters in Intel i82599 NIC: The Intel i82599 NIC provides a wide range of different filters, each with different configuration options. Two examples of these filters are *5-tuple filters*, and *flow director filters*. The *5-tuple filters* are specified by a 5-tuple that includes (i) type of protocol, (ii) source IP address, (iii) destination IP address, (iv) source port, (v) and destination port . Any field in the tuple can be masked to provide the wildcard matching behavior so that the filter can match any value for the masked fields. The *flow director filters* can either be configured to provide a hash-based imperfect matching or an exact matching on the configured set of fields in the 5-tuples.

We present an example of modeling 5-tuple filters in the Fig. 3.4. The Unicorn code for the `C5TupleFilter` c-node captures the configuration space of this filter with the keyword **type**. This composite type tells the exact format of the configuration values and the valid range that can be configured in the filter.

The Unicorn code also provides a Haskell function `config5tuple` which models how a given configuration value will change the c-node. We will revisit how the configuration values are implemented by the typical filter c-nodes in greater details in the later chapter (5.3.1).

In Fig. 3.5c, we show an example with two filters. The first 5-tuple filter is configured to match UDP packets with a destination port equal to `7` (shown with f-node `5T(UDP, *, *, *, 7)` and forward them to the `RxQueue1` hardware queue. The second 5-tuple filter is configured to match UDP packets with a destination port equal to `11211` (shown with f-node `5T(UDP, *, *, *, 11211)`) and forward them to the `RxQueue2` hardware queue.

We further discuss the filter configuration management (section 5.3.1) in chapter 5 in the context of hardware queue management.

3.4 Exploring the configuration space

A model of the hardware capabilities provided by the PRG is a first step towards a portable resource management solution. The next step is to find a configuration which is relevant for the current network stack state (captured by LPG). This is done by exploring the configuration space of the PRG.

The exploration of the configuration space is hard because the behavior of the NIC depends on the configured values in all the configurable fields. We can't evaluate configuration of a one field at a time. Instead, we need to generate values for all the configurable fields in the NIC, and only then we can determine the behavior of the NIC under that configuration.

For example, the PRG shown in Fig. 3.5c has a relevant configuration for the LPG described in Fig. 3.1 with two flows (`5T(UDP, *, *, *, 7)` and `5T(UDP, *, *, *, 11211)`) mapped into two hardware queues (Q1 and Q2 respectively). Even though configuring two filters is sufficient to steer the flows in proper queues, we still need to consider the configuration values of all 128 filters to check if any other filter is also steering packets to these two queues. This is simple example of how the behavior of NIC depends on the configuration of entire NIC.

A naive approach would be to walk through the entire configuration space of the PRG and test if the current configuration maximizes the benefits based on the current resource allocation policy.

The naive approach of evaluating all possible configurations is prohibitively expensive as the configuration space of modern NIC is huge. For example, the Intel i82599 NIC supports RX packet filtering based on five tuples in the `IPv4` packet header. This involves configuring both source/destination `IPv4` addresses and source/destination port numbers, port type and a destination queue. These values can be configured in $2^{32} * 2^{32} * 2^{16} * 2^{16} * 2^{7}$ different ways. The i82599 NIC has 8K such flow director filters, increasing the configuration space further to approximately 2^{116} bits just to configure the flow director filters. This is not the only type of filter as this NIC also supports 32K signature based filters, 128 5-tuple filters, Ethertype filters, etc. We discuss these filters in further details in later chapter (5.2.2). The filter configuration is just one type of configuration. There are other capabilities (e.g., virtualization, rate control) which can be configured. Thus leading to a further increase in the configuration space. Hence, we believe that a comprehensive search of the configuration space would require an unreasonable amount of time.

We can prune a large part of the configuration space by using information from the LPG about the current network state. We can, for instance, limit the PRG configurations based on the current active flows and protocols used. This approach of using the current network state to limit the configuration space depends on understanding which hardware features are relevant for which network state, and how to configure a relevant hardware feature based on a given network state. For example, to generate a configuration for U-PRG (Fig. 3.5a) based on the network state in the LPG shown in Fig. 3.1 to reach the state shown in C-PRG (Fig. 3.5c), we need the knowledge that the flows wildcards (e.g., `5T(UDP, *, *, *, 7)` and `5T(UDP, *, *, *, 11211)`) can be mapped onto the 5-tuple filters by applying the appropriate mask.

We take the approach of using the network state for exploration of configuration by implementing NIC specific *oracles*. This oracle can use the hardware-specific knowledge about the relevance of a hardware feature for the network state to propose configurations which are meaningful to the network state. Hence this approach significantly reduces the configuration

space.

3.4.1 Hardware oracle

The hardware oracle tackles the problem of large configuration space exploration by only suggesting reasonable configurations which are then further evaluated by the search step (3.6.2). The oracle takes the hardware PRG, current hardware configuration, and the requested change in a network state as an input. It uses this information and the NIC-specific knowledge to suggest a set of incremental changes to the given configuration.

The oracle is responsible for returning the few configuration changes which are most relevant to the requested network state change in the current situation of the NIC configuration and the current network state. Another way of thinking about the hardware oracles is that they are responsible for mapping the network state onto the NIC hardware capabilities so that we do not have to struggle blindly with all possible configurations.

The oracle is basically an optimization to reduce the search space of potential configurations. For example, the most extreme version of an oracle is the *generic oracle* implementation that returns all possible configurations irrespective of the network state. The other extreme is the *minimal oracle* implementation which always returns a basic configuration needed to make the NIC functional without using any additional features, or the *hardcoded oracle* which always return a single fixed configuration.

Oracle developers need to understand the network state and its implications on the NIC hardware configurations. They are responsible for the trade-off between the flexibility of using NIC hardware features and reducing the search space. The hardware oracle needs to limit the way in which a NIC can be used, but it should not implement a resource management policy. For example, the *minimal oracle* enforce a policy of using no additional hardware features, and the *hardcoded oracle* enforces a single policy. The *generic oracle* allows any configuration but at the cost of larger search space exploration. We discus an implementation of the hardware oracle in further details in the later chapter (5.3.2).

3.5 Understanding application requirements

A network stack needs to understand what exactly the application is trying to achieve with network flows, so that it can provide those functionalities while exploiting the NIC hardware capabilities whenever applicable.

As most old NIC hardware was limited to the simple transmitting and receiving of the packets, traditional network stacks provided a simple interface to applications to facilitate requesting flows and for data transmission and receiving. A common interface for this is the socket interface [LJFK86].

Even though the socket interface has evolved to support additional attributes flows using `setsockopt` [soc08], the flexibility provided to applications for fine-grained control over flow manipulation is limited. For example, it is difficult for applications to control load balancing of the incoming traffic on a single socket between multiple application threads running on separate cores.

Moreover, the socket interface makes it difficult to exploit NIC hardware capabilities for the benefit of applications, and therefore researchers have started exploring ways to extend the socket interface to take advantage of the hardware capabilities [JJ09].

The socket interface is also restrictive when the NIC hardware implements partial application logic (e.g., ApplicationOnload Engine [Sol12]) as it does not provide any means to specify which application processing can be pushed to the hardware. To accomplish offloading, the applications need to rely on vendor-specific interfaces.

We need a more flexible interface to capture the application requirements for packet processing so that applications can benefit from the increasing NIC hardware capabilities without having to rely on vendor-specific interfaces.

3.5.1 Interfaces based on dataflow model

Ideally, the application should be able to specify its requirements using a detailed dataflow graph including the desired flows, packet processing, and

other application specific requirements. Using the flexible interface of a dataflow model can allow applications to offload application level packet processing. For example, an application can specify which part of the packet payload it is interested in, and which packets can be dropped without delivering to the application.

Even though the dataflow-based interface can be very powerful, we are currently not targeting hardware which supports the offloading of application logic. Hence, we have settled on a simpler, network flow based interface in our current implementation (presented in the next section 3.5.2). We use this interface to collect application requirements and use them to update the LPG.

We believe that extending Dragonet to directly expose the generic dataflow-based interface to applications should be feasible with some more engineering. We are already using the dataflow model for most of our processing, and exposing it directly to the applications should not be difficult.

In addition to not exposing the dataflow-based interface to applications directly, we are currently also missing a mechanism which applications can use to precisely and safely define the processing required by them. These two extensions will be needed in the current Dragonet implementation to exploit fully programmable NIC devices.

3.5.2 Network flow management interface

Dragonet network stack provides a network-flow-based interface to the applications which can be used to register for individual network flows, or to manipulate them.

We define a flow as a predicate on a network packet. For example, a listening UDP socket defines the class of packets that will reach the socket. Similarly, a TCP connection defines the class of packets that are part of this connection. It is worth noting that, even though UDP is connectionless, we can still define a UDP flow as the class of packets that have a specific source and destination UDP endpoints (IP/port). The predicate can be considered as a tuple of the protocol types and the protocol-specific addresses of the flow endpoints. In addition, any of these fields can be partially or fully

masked for a more inclusive definition of a flow, or to group certain flows. This representation maps well to the address abstractions used by the socket interface for flow-manipulating calls (e.g., connect and bind calls).

Typical network flow operations include registering or closing flows, dividing them between multiple endpoints, or controlling the way a flow will be forwarded. Once the network flow is assigned to the application, the application can directly send and receive data on it using the data plane interfaces.

In addition to requesting a flow to be directed to it, an application can also specify the desired attributes for the flows. These attributes can be priority, bandwidth or latency requirements, or load balancing policies.

The *network flow interface* differs from the socket interface as it only deals with the control plane, whereas the socket interface provides both the control plane and the data plane interface. Regarding flexibility in manipulating the flows, the *network flow interface* is located between the socket interface and the dataflow-based interface. It supports more manipulations and flow attributes than the socket interface, but it does not provide mechanisms pushing down application logic.

Dragonet uses the flow definitions and the attributes to capture the application requirements and populate the LPG with them. In addition to specifying how the network flows should be forwarded to the applications, this LPG is then used for policy-based management of NIC capabilities.

3.6 Policy-based resource allocation

After generating a set of interesting configurations using the hardware oracle, we need to select one configuration we will use. This selection gives us an opportunity to enforce certain resource utilization policies by making sure that the selected configuration meets the criteria needed by the resource allocation policy. We formulate this as a search problem where our goal is to find a configuration that is most suited for current resource allocation policies. Fig. 3.6 gives an overview of the steps involved in the policy-based resource allocation.

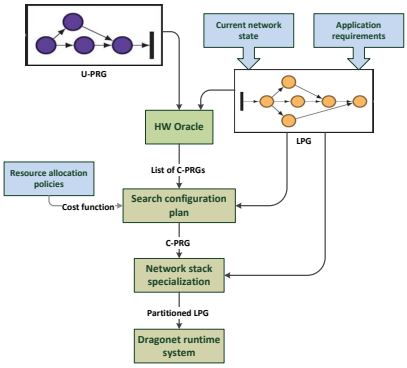


Figure 3.6: High-level view of the steps involved in policy-based resource allocation using Dragonet

A naive approach to resource allocation is to hardcode the policies and criteria directly in the search process. Different resource allocation policies can be implemented by selecting different search implementations. We take the generic approach of separating the policies using a **cost function** that will be used to score the configurations during the search. This separation helps us to simplify the implementation of policies as a cost function and enables us to test different search strategies without worrying about the policies.

This design decision also implies that we will not be able to reduce the search space by exploiting policy-specific properties, but we believe that the simplicity and flexibility provided by this separation are worth losing a few potential optimizations.

We discuss the search problem next (3.6.2) and then present the cost functions (3.6.1)

3.6.1 Cost functions: Hardware agnostic policies

We need a way to decide how "good" a given configuration is so that we can select a useful configuration in a hardware-agnostic way. As the definition of "goodness" depends on what application developers and system administrators want to achieve, we need a mechanism by which the administrator

can specify their goals. We directly use the definition of "goodness" as the NIC resource management policy, and we use a **cost function** abstraction for defining the goodness for a desired resource allocation plan. The cost function provides a way to score any resource allocation plan given to it based on a user-defined policy and hence enables searching for the least costly configuration.

As the cost functions represent resource allocation policies, they are supposed to be **hardware-agnostic** and should work with different NICs. This requirement implies that cost functions should only operate on the NIC independent models to evaluate the resource allocation plans. Our initial plan was to perform the cost function evaluation using an **embedded LPG**. The embedded LPG essentially models how the LPG (capturing network state and current requirements) can utilize the capabilities provided by the C-PRG (capturing a particular hardware configuration). We will discuss the embedding approach and the creation of embedded LPG in more details in the next section (3.7.2). Using the embedded LPG works well when the resource allocation plan does not have to be frequently reviewed. However, when the resource allocation plan needs to be reviewed frequently, performing embedding step for each potential hardware configuration suggested by the hardware oracle, during each review of the resource allocation plan is prohibitively expensive.

We have simplified the problem above by making an assumption that most of the control-flow related interactions of an application with a network stack are related to network flows. We use this assumption to optimize the cost function evaluation step by using a *Qmap* representation (discussed in later chapter 5.4.3) which is centered around the network flows and its attributes (e.g., groupings, bandwidth requirements, priorities).

An administrator writing a policy using a cost function is expected to evaluate the given *embedded LPG* or the *Qmap*, and based on how desirable the given resource allocation plan is, the policy developer can score with a numerical value. Typically, this can be done by calculating a distance from the ideal resource allocation plan.

One example of a simple policy is *to use the NIC hardware for computations whenever possible* and hence reducing the load on a CPU. This policy can be easily implemented with a cost function which counts the number of the

software nodes in the embedded LPG to find the NIC configuration with least CPU overhead.

In addition to the numerical value representing a non-negative cost (zero cost representing the least expensive resource allocation plan), we allow the cost functions to report a verdict on the given resource allocation plan. The verdict can be one of following three:

- `CostReject` verdict implies that this configuration is not acceptable.
- `CostAccept` verdict implies that this configuration meets all the requirements of the policy, and there is no need to keep searching further.
- `CostOK` verdict says that the configuration meets few of the requirements of the policy, but there is a scope for further improvement.

These verdicts reported by the cost functions help the search to converge on an acceptable solution quickly. The use of these verdicts is an optimization, and a cost function can always choose to report `CostOK` with a numerical score. In this case, a search will have to check all configurations taking more iterations.

Our approach of using cost functions allows the composing of simpler or application-specific cost functions to create more complex or system-wide cost functions. Composite cost functions can be developed by combining the output of simpler cost functions with either a decision tree or by weighting them.

We will give examples of a few cost functions in later chapter (5.4.4).

3.6.2 Searching the PRG configuration space

The search phase is responsible for exploring the configuration space of the NIC by using the hardware oracle to generate the relevant configurations and then using the cost function to evaluate them. The goal of a search function is to identify a configuration which will minimize the cost. The search step determines the quality of the resource allocation by controlling how frequently and how deep the configuration space will be explored.

In other words, we can map the resource management problem into a search problem by using the cost functions to evaluate the NIC configuration (captured by the C-PRG) in the context of the current network state and application requirements (captured by the LPG).

The quality of the resource management depends on how often the search for the least expensive configuration is performed, exposing the trade-off between added overhead of the search and the reconfiguration of the runtime system with the freshness and the quality of the resource management plan. At one extreme, we can invoke the search for every application request, or whenever a new flow detected in the system. This approach will provide the ideal hardware configuration in all cases, but at the cost of performing the full search and the reconfiguration of the runtime system for every change, which may not scale for a system with a large number of flows. On the other hand, we can carry out the search after a fixed interval or after a certain number of changes observed, which will amortize the search and reconfiguration cost over these changes. We take an approach of allowing applications to request explicitly for the search for the configuration space as added optimization.

We further discuss the implementation of the search in later chapter (5.4.5).

3.7 Dataflow-based Interfaces

Once the NIC hardware is configured, we need an interface to pass the information regarding the configured hardware functionality to the network stack so that it can use the information to exploit the work done by the NIC hardware. If this interface is hardware dependent, and understanding the information provided by the interface requires hardware specific knowledge, then using hardware capabilities in a portable way becomes difficult.

A traditional network device driver layer typically provides a queue-based interface for transmitting and receiving packets [LJFK86]. In Linux, the device driver interface also provides an `IOCTL` based mechanism [RC01] which are used by `ethtool` [eth] to query and configure the NIC hardware features. This interface pushes the responsibility of understanding the

semantics of the hardware configuration and its implications to the applications or to a system administrator.

In the related work section (2.3.2), we discussed few alternate approaches which re-visited this interface albeit with different motivation. In this section, we present the **C-PRG** interface, which is based on a dataflow model. The motivation for this interface is to provide the information about currently configured NIC hardware functionalities in a hardware agnostic way.

3.7.1 Interface based on the C-PRG

Dragonet uses the configured PRG (C-PRG) as an interface to provide information about the NIC hardware capabilities to the network stack. As a C-PRG only contains f-nodes and o-nodes, it is hardware agnostic and captures packet processing done by the NIC hardware. This PRG can be analyzed by the network stack to infer the attributes and invariants which will hold on all endpoints of the NIC (e.g. RX and TX queues).

This interface exposes a trade-off between the ability to use the configured hardware functionalities and the complexities of analyzing and adapting to a *configured PRG*. At the one end, the network stack is free to ignore this information and use the queue-based interface provided by PRG endpoints without worrying about the work already done by the hardware. At the other end, the network stack can utilize this information by doing additional work of analyzing the *configured PRG* to figure out which processing it can avoid, and adapting to prevent the duplicate work. In the worse case where no processing can be avoided, the network stack needs to do all the work in the software.

Limitations of C-PRG Interface

The configured PRG depends on the PRG developers encoding enough information in the PRG which can be used by the network stack. This requirement is reasonable because it is a one-time effort and provides a detailed PRG leads to a better utilization of the NIC hardware. Hence it is in the best interest of a NIC vendor to provide detailed PRGs for their hardware.

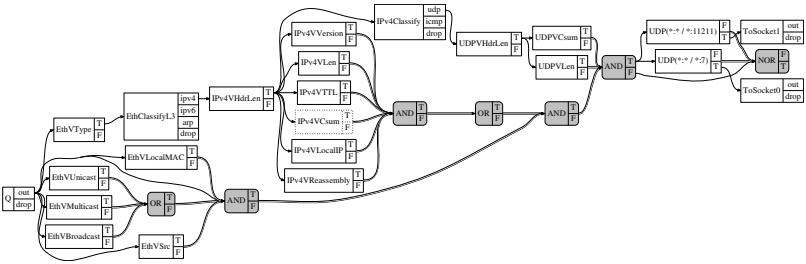


Figure 3.7: Software nodes of an embedded graph that results from embedding the the LPG shown in Fig. 3.1 on the default RX queue `Q0` of the C-PRG shown in Fig. 3.5b. The `EthClassified`, `EthVCRC` and `EthVLen` nodes are offloaded on the PRG, and the `IPV4Classified` node is imported from PRG to meet the software dependencies of PRG.

However, this interface is not completely future-proof, as this interface can lead to situations where a network stack does not understand the detailed attributes provided by the PRG. In such situations, a network stack should be in a position to safely ignore these additional attributes and still work correctly by using only the information it understands. For example, if a network stack understands flow classification constraints but does not understand bandwidth controlling attributes provided by the PRG, then it can still make assumptions about where the flows will end up.

We rely on a dataflow model to ensure that dependencies are met and that there are no conflicts between different types of attributes. Each attribute can be independently analyzed, and the assumptions made on them can be aggregated. This independence of the attributes helps us with making the robust.

3.7.2 Embedding using node labels

One way of benefiting from the information provided by C-PRG is to avoid duplicating the work that the NIC hardware is already doing by **embedding** the LPG graph onto the C-PRG. The embedding process should remove all functionalities from the LPG which are already being performed in the PRG.

Our initial approach is to use the labels on f-nodes to compare the functionalities of the nodes in PRG and LPG as a basis for working out which protocol processing is already done in the NIC hardware. Simple protocol processing, which can be either turned on or off, can be easily offloaded using this approach. We implement this simple algorithm by finding the f-nodes which are common in both PRG and LPG and either move them into the LPG if the PRG node is implemented in software, or remove them from the LPG if the PRG node is implemented in hardware.

The result of this simple embedding algorithm is shown in Fig. 3.7. This example shows that our embedding approach can easily offload basic functionalities like CRC checksum (node `EthVCRC`), validating packet length (node `EthVLen`) on the PRG. The example also demonstrates that our simple algorithm can correctly handle the PRG software nodes by moving them into the embedded graph (e.g. `IPV4Classified`).

To generate a fully embedded graph, we first perform the simple embedding and then attach the embedded LPG to each active RX queue of the NIC ensuring that the full packet processing will be executed on all the packets irrespective of the RX queue on which they arrived.

Limitations of label-based embedding: The label-based embedding approach works well for simple hardware capabilities which can be turned on or off but the approach is difficult to scale with complex configurations. For example, the Intel i82599 NIC supports wildcard entries for the 5-tuple receive-side filter configurations. Such a flexibility makes it difficult to perform the matching of LPG flow classification nodes and PRG filters just by using the node labels. On top of that, the final queue where the packet will arrive, depends not only on the filter which matched the packet, but on all previous filters which missed that packet. This behavior makes the label-based node matching of LPG flow filtering node with PRG configured filters difficult.

```
graph example {
    boolean L2EtherClassified {
        port true[ValidLength]
        port false[]
        semantics true { (= L2P.Ethernet (L2.Proto pkt)) }
    }

    node ClassifyL3 {
        port ipv4[.L3IPv4ValidHeaderLength]
        port ipv6[]
        port arp[.L3ARPValidHeaderLength]
        port drop[]
        semantics ipv4 { (= L3P.IP4 (L3.Proto pkt)) }
        semantics ipv6 { (= L3P.IP6 (L3.Proto pkt)) }
        semantics arp { (= L3P.ARP (L3.Proto pkt)) }
    }
}
```

Figure 3.8: Unicorn snippet showing an example node with boolean predicates added using a semantics attribute

3.7.3 Using predicates for flow analysis

To simplify the embedding of complex functionalities in our model, we use the approach of associating the outgoing ports on f-nodes with predicates, and then we use constraint matching. For every f-node, we add attributes specifying a predicate which will be applied when a packet is forwarded to the particular outgoing port.

The predicates make it easy to reason about the state of the packet when it reaches the particular f-node. We create a boolean predicate encoding the condition of a packet by applying predicates from the selected ports of all the traversed nodes. We do same with the nodes in LPG to get a boolean predicate required by the network state. These two predicates can then be compared to decide if a given LPG node can be embedded or not, and to determine the additional predicate required to complete the functionality required by the network stack. We further discuss this approach in chapter 5 in the context of *Mapping flows into queues* (5.4.3).

This approach provides certain guarantees on the packets reaching a particular RX queue in the form of aggregated predicates, and these guarantees

can be used by the network stack to create specialized data paths which can implement optimizations based on these assurances.

3.8 Conclusion

We have explored the use of dataflow models to capture the capability of the NIC hardware and the state of network processing such that NIC hardware capability can be effectively used. We have shown that the simple model using two building blocks (function nodes and operator nodes) can capture the state of the network stack, and the c-nodes can be used to model the configuration space of the NIC hardware.

We have discussed the difficulties in searching the configuration space of the NIC hardware, and proposed reducing the configuration space by using a hardware specific oracle to provide the configurations which are relevant to the current network state.

Dragonet provides a way to systematically explore the NIC configuration, and this exploration capability further enables us to select NIC configurations based on high-level resource management policies. We have explored interfaces based on cost functions to implement these policies in a hardware-agnostic way.

Just configuring the NIC hardware is not enough to take full advantage of the NIC features. In addition, we need a way to share information regarding the currently configured hardware capabilities with the network stack. We have used configured PRG (C-PRG) as a hardware-agnostic and flexible interface to share the information about the NIC processing capabilities with a network stack.

In this chapter, we have presented the design of the Dragonet network stack and how it can provide information about the hardware capabilities in a systematic way. The next chapter will discuss how this information can be used to create and run efficient network stack.

Chapter 4

Dragonet runtime system

The Dragonet provides information about the protocol processing capabilities of NIC hardware by using hardware-agnostic dataflow interfaces. In this chapter we discuss how we can use the information about the NIC capabilities to generate a specialized network stack which is tailored for current network state and can benefit from the currently configured hardware capabilities.

We also describe the execution framework we built to implement a host network stack prototype using information provided by the dataflow model to determine the required protocol processing. This execution framework provides a runtime which can adapt the protocol processing based on the changing application requirements or hardware configuration.

4.1 Background

The idea of a specialized network stack has been around for a while and has been used with various motivations.

The *exokernel* [EKO95] approach uses a specialized user space network stack which optimizes protocol processing based on the application logic to

improve the application performance. The overhead of generalized network stacks and the performance benefits of user space network stacks which can exploit application semantics are also studied in the *Sandstorm* [MWH14] network stack.

The highly configurable *CiAO/IP* network stack [BLS12] uses the ideas from *aspect-oriented programming* to specialize the stack for embedded systems by removing unwanted functionalities, hence saving space.

The *Unikernel* [MS13] approach of *MirageOS* uses a high-level programming language for the implementation of the OS services, the network stack and the application logic which can directly run on a VM. It utilizes the language framework to perform compile-time safety checks and optimizations (e.g., dead code elimination) on the entire stack including applications, network stack, and OS services.

The Dragonet approach focuses on the benefits of the capabilities of the NIC hardware by specializing the network stack based on the guarantees given by the current configuration of the NIC hardware and the application requirements.

4.2 Creating a specialized network stack

The previous chapter 3 has introduced the dataflow interface called Configured-PRG (C-PRG) in section 3.7.1 which captures the packet processing capabilities of the currently configured NIC in a hardware-agnostic way. It also presented Logical Protocol Graph (LPG) as a way to capture the current state of the network stack in section 3.3.2. Here, we use the information about the network stack state from LPG and the information about the hardware capabilities from C-PRG to create a specialized network stack optimized for current network state and hardware capabilities.

The information available in the C-PRG and LPG needs to be used by the network stack to benefit from it. The exact approach for using this information depends on the goal and the priorities of the network stack. Here we present a way to create a specialized network stack which is optimized to perform well on a multicore CPU system by partitioning the execution of

the protocol graph into separate components with the help of information provided by the C-PRG. These components can be deployed on different cores to exploit the available parallelism between different flows.

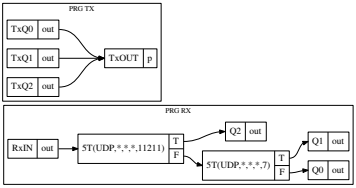
As we are interested in the multicore performance of a network stack, we currently limit our embedding algorithm to use the hardware queues and filters for partitioning the execution across multiple cores, and ignore other offloading features.

Fig. 4.1 illustrates how a network stack can be specialized by creating customized protocol processing components for each application based on the application requirements, and the functionalities provided by the hardware endpoints. We use a simplified LPG (Fig. 4.1b) based on the LPG described in Fig. 3.1 with two applications each listening on one UDP port (port 7 and port 11211). For the PRG, we use a simplified C-PRG (Fig. 4.1a), which is based on the PRG presented in Fig. 3.5c configured with two hardware filters.

The process of specializing the network stack starts with connecting a copy of the LPG with each pair of send and receive hardware queues with the same IDs as shown in Fig. 4.1c. This step ensures that the complete packet processing will be applied to every packet irrespective of which NIC endpoint the packet may arrive.

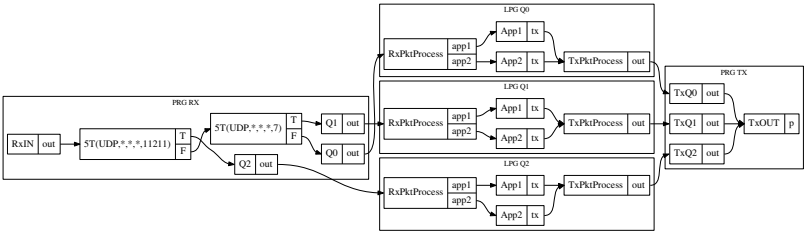
Next the specialization process simplifies the embedded LPG by removing the parts of the graph which can't be reached by the packets (Fig. 4.1d). This step relies on information from the configured PRG to work out which packets will be demultiplexed on which hardware queue. Using this information for each RX queue, this step removes unreachable nodes from the LPG connected with that RX queue.

The reduction step depends on the semantic information associated with ports of the nodes specifying conditions on the packets traversing that port. This step creates an expression for each port by aggregating the predicates from the predecessor ports which a packet may have to traverse to reach the current port. The expressions for each port will be checked for satisfiability using an Satisfiability Modulo Theories (SMT) solver. If the expression is unsatisfiable, then it is assumed that a packet will never traverse that edge and that particular edge can by safely dropped. Once the edges with unsat-

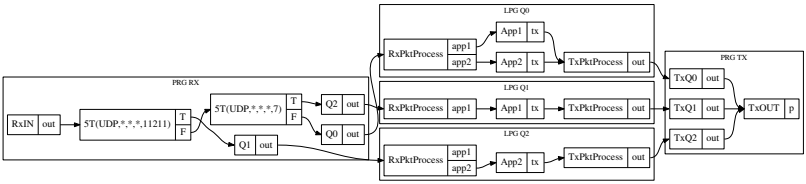


(a) Simple PRG

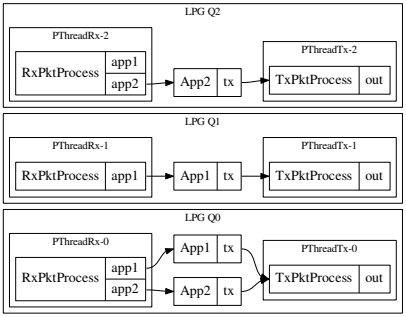
(b) Simple LPG



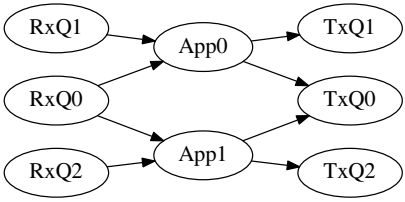
(c) Step 1: Connecting LPG instance to each PRG queue-pair



(d) Step 2: Simplifying the Embedded LPG



(e) Step 3: Partitioning the LPG



(f) Deployment components with explicit communication

Figure 4.1: Steps involved in creating a customized Protocol processing

isfiable predicates are dropped, then the nodes which are not reachable from the starting node can also be dropped. We have implemented the necessary boolean predicate logic using a custom solver that we developed in Haskell. We also experimented with the Z3 SMT solver [DMB08]. There is significant room for further optimizations and performance improvements in our current approach of using boolean solver (e.g., use faster solvers, transform the problem into simpler problems) and we haven't fully explored different options for performance improvements yet.

In our example, the C-PRG from Fig. 4.1a assures that `Q1` will only receive UDP packets with the destination port `7`, and hence the LPG instance connected to it can be simplified by removing the processing of all other protocols and flows, except for the processing required by the flows of `App1`. Similarly, the LPG instance connected to `Q1` can be simplified by only keeping the processing needed for the flows of `App2`, leading to the simplified embedded LPG shown in Fig. 4.1d.

The protocol processing connected with the default receive queue `Q0` is not simplified to ensure correctness. We take this approach to handle corner cases like handling fragmented `IPv4` packets or handling delays in updating the hardware configuration.

The reduction step of removing unreachable edges and nodes is an optimization, and is not necessary for the correctness of the packet processing captured by the embedded graph. The only difference is that the reduced graph may have improved performance due to the simplified protocol processing.

Next, the specialization process will prepare the embedded LPG graph for partition by removing the hardware nodes; it is done by finding the RX and TX queues and deleting all the nodes before and after them respectively.

4.2.1 Creating partitions

Next, the network stack specialization process will partition the embedded graph into separate components to meet our security requirements and facilitate deployment on multiple cores.

Each component is assumed to be a self-contained protocol processing unit with explicit communication channels to other components. These components can be deployed on separate cores with their own execution engine (discussed in section 4.3.1).

Currently, to provide better locality, we have taken the approach of creating a separate component for each hardware queue-pair, to make sure that the packets arriving on the RX queue are processed by the same partition. These partitions can be deployed on different cores. The partitioning step in Fig. 4.1e shows three separate components, one for each hardware queue-pair.

In addition to the performance implications, partitioning step is also used to handle the security constraints implied by having direct access to the NIC hardware queues. Depending on the capability and configuration of the NIC hardware, the verification of the addresses of the packet buffers provided to the NIC hardware queues might be necessary. Ideally, the C-PRG model should be able to provide enough information to the network stack regarding this, but our current implementation does not include these attributes. We have taken a conservative approach of not trusting the applications with direct access to the NIC hardware queues. This conservative approach is implemented by moving the trusted queue endpoints into the separate protocol processing components (shown with `PThread*` in Fig. 4.1e) during the partitioning step.

Our partitioning algorithm makes the communication explicit by grouping the edges which are crossing components and providing explicit communication channels for sending the packets over these cross-partition edges by inserting additional nodes. These additional nodes are inserted at the beginning and end of each component. They are responsible for multiplexing/demultiplexing the cross-partition edges and enqueuing/dequeuing packet buffers in the communication channel. The explicit communication channels needed in our example are shown in Fig. 4.1f.

The implementation of these special nodes is discussed in the next chapter as part of the execution engine (4.3.1). The implementation of the communication channel uses a shared memory based single producer-consumer bulk transport library and we discuss the implementation details in the next chapter (4.3.1).

Our approach of partitioning the embedded graph makes the placement of each node and the communication between the partitions explicit. This information is useful in resource planning based on how much communication and computation is needed for the deployment of the partitioned protocol graph.

4.2.2 Alternate approaches

We use the C-PRG to create a detailed execution plan in the form of a partitioned LPG, optimized for deployment on multiple cores. The reader should note that our partitioning implementation shows one way of using the information provided by the C-PRG for the particular optimization goal; this information can be used in different ways based on different optimization goals and the security constraints for the deployment of the network stack. For example, if the security constraints on the deployment are relaxed by fully trusting the application, then full protocol processing, including hardware queue access, can be pushed into the address space of the trusted application. Such deployments will be equivalent to zero copy user space networking and can avoid the overhead of additional partitions and communication channels.

Next, we discuss how our task-based runtime system uses the partitioned LPG to deploy the customized network stack.

4.3 Runtime system

The motivation for the Dragonet runtime system is to provide a custom **control plane** and **data plane** to assemble the network stack based on the hardware capabilities and the application requirements. We take the approach of using the dataflow model directly for our packet processing. We implement an execution engine which can directly run our dataflow model for packet processing. Our control plane adapts with changes in the resource allocation plan by directly updating the dataflow models used by the data plane execution engines.

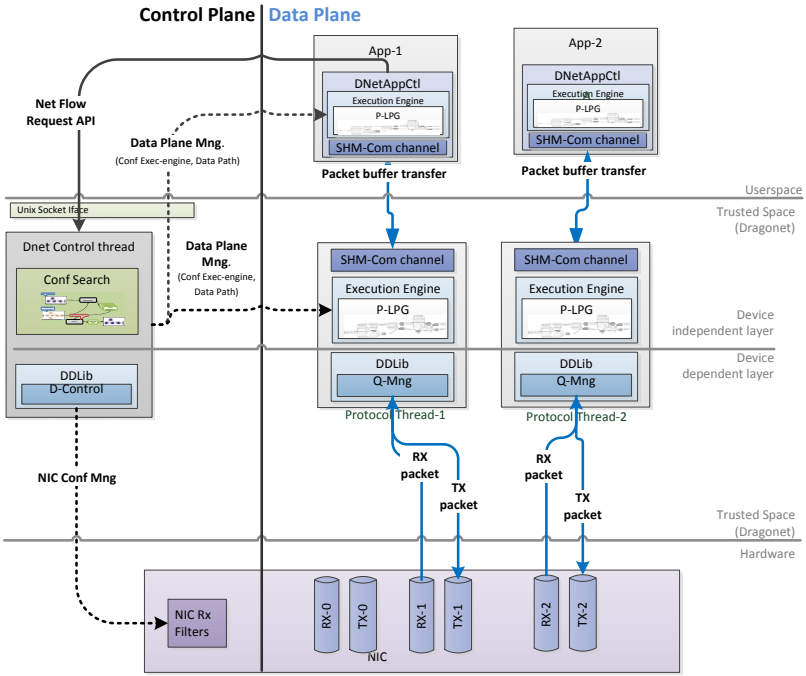


Figure 4.2: Overview of the Dragonet runtime system

Fig. 4.2 gives the overview of the Dragonet runtime system. We divide our runtime system in the **control plane** (shown on the left side of the figure) and the **data plane** (shown on the right side of the figure). The data plane (4.3.1) is responsible for moving packets between applications and NIC using execution engines, communication channels and the device-dependent queue interface. The control plane (4.3.2) is responsible for managing the data planes by setting up and orchestrating them by managing the execution engines, communication channels and the NIC configuration. The applications interact with the Dragonet runtime system by using the application interface (4.3.4).

4.3.1 Data Plane

The primary goal of our data plane is to implement the packet processing specified by the partitioned LPG (P-LPG) efficiently. To facilitate packet processing based on the partitioned LPG model, we define an execution model which specifies how to interpret and use the LPG for packet processing. We first explain our initial pure dataflow model and the issues we encountered with it; then we describe our extended task-based dataflow execution model which resolves these issues, with its implementation. We describe the implementation of shared-memory-based bulk transport channels for providing communication between different execution engines and the packet transfer.

Pure dataflow model

The pure dataflow model is a simple and intuitive model which works by creating an *acyclic directed graph* from the f-nodes and o-nodes of the LPG and executing it on each packet to provide complete packet processing.

The LPG in this model can be interpreted in two ways. In the forward direction the edges represent the execution order, implying that once the execution of a given node is complete, any of the successor node selected by the enabled port can be executed. In the backward direction the edges represent dependencies between nodes, implying that every node is dependent on the execution of all the predecessor nodes.

Execution of the model: The representation of the LPG in a pure dataflow model can be used to perform all the packet processing needed by the current state of the network stack. This processing is done by first finding the entry nodes (nodes without any incoming edges), and executing them on a given packet buffer. The processing of each node will enable one of its outgoing ports and hence adding the function nodes pointed by the enabled port into the execution set. The operator nodes are special in the packet processing as its processing depends on all of its incoming edges. The packet processing will continue with the nodes in the execution set, and the packet will be completely processed when there are no elements left in the execution set.

Limitations of the pure dataflow model: Even though the pure dataflow model is sufficient to capture basic packet processing, it shows limitations in capturing the non-deterministic behavior where a single f-node execution can trigger multiple executions of the same execution path. One example of such processing is handling ARP replies. When a single ARP reply arrives, more than one packet may have to be processed as a response, based on the number of packets currently waiting for that particular lookup.

Task-based model

We extended our pure dataflow model to a task-based model to address its limited support for non-determinism. We introduced **spawn edges** to capture the capability of a node to start non-deterministic tasks. A spawn edge originates from a function node and points to a destination node which it can start as a separate task. During the execution, a node can use the spawn edge to start the arbitrary number of tasks pointed by the spawn edge.

Execution of the task-based model: This extension also changes the way our model can be executed. The new model uses a *task queue* on top of the packet processing used by the previous model. Each task in a task queue is a closure of an f-node and an optional packet buffer. The f-node is used to start the execution, and this f-node can use the packet buffer provided. Each task is then executed same as a pure dataflow model described earlier, with an exception that a node with a spawn edge is allowed to push new tasks in a task queue. This approach of allowing a function node to create additional packet processing tasks enables us to overcome the limitations of the pure dataflow model in handling non-deterministic situations, such as processing ARP replies.

We will discuss the implementation of the task-based execution engine next.

Implementation of the execution engine

The execution engine implements a network stack by running a protocol graph provided by the Dragonet control thread. We are using a task-based

execution engine which works by maintaining a task queue, and executing each task to completion. Each task is comprised of a starting function node in a protocol graph and an optional buffer.

The execution engine will pick a task from the task queue, and start the execution with the first function node. The function node is responsible for implementing the logic and returns which port should be enabled. The execution engine uses the returned port to determine which nodes need to be executed, and adds them to the execution set.

The nodes in the protocol graph are implemented as functions in the C language for efficiency purpose. These functions take parameters for node context, global state, and buffer handle and return a port identifier based on which port should be enabled.

The global state is used to maintain the protocol specific information that will be required by more than one node in a protocol graph. The address of a local endpoint is an example of the information which will be required by more than one protocol graph node.

The buffer handle includes a buffer where the packet is stored and the list of attributes associated with the packet. The attributes can contain information like packet length, header length, payload offset, etc. These attributes can be viewed as a way to pass information about the packet to the successor nodes in the protocol processing.

Implementation of f-nodes: A function node is the most common node and is responsible for most of the packet processing. A function node is allowed to access and modify the global state, the buffer handle, and the buffer containing the packet and its attributes. A function node can allocate a buffer if there is no valid buffer in the buffer handle, and can free the buffer. These buffer management capabilities are typically used by initial and terminal nodes.

A function node can also start new tasks if it has a spawn edge originating from it by submitting a spawn operation to the current execution system. This request includes a node context, a spawn edge identifier, priority of the task, and an optional buffer. The spawn operation can be submitted multiple

times from a single execution of the function node. The spawn operation instructs the execution system to create a new task with the node pointed by the spawn edge as a start node and the buffer handle provided in the spawn request. This task is then added at top or bottom of the task queue based on the priority specified in the spawn request.

Implementation of Operator nodes: The operator nodes in the Dragonet model can be seen as synchronization mechanisms where several packet processing flows are joined, and based on the results of the input processing flows, the output flow is enabled. The implementation of these operator nodes is more complicated as they can be enabled multiple times due to their multiple inputs but the successor nodes should be executed only once. We also support short-circuit semantics by enabling the output ports if enough inputs are enabled to make a decision.

We implement the operator nodes as part of the execution engine which keeps track of the enabled input edges and performs the boolean operations on them. The execution engine maintains the output status and the version number based on the iteration number for each logical operator. It uses this information to determine if the operator node is already enabled for the current iteration, and hence does not need to be processed again.

Implementation of the device dependent nodes: An execution engine may have to deal with NIC specific nodes as these nodes can get pulled into the embedded LPG as part of the embedding process. These hardware dependent nodes are typically responsible for interfacing with the NIC for send/receive packet operations. They are also responsible for implementing the software component of the functionalities which are partially implemented by the NIC hardware.

Example of a device dependent node is a node which interprets the result of the checksum calculation performed by the Intel i82599 on incoming packets. The i82599 NIC stores the results into the packet descriptor in a device-specific format. We have modeled this hardware capability with the node `IPv4Csum_` in our example PRG in Fig. 3.5. In order to use the results of this checksum calculation in LPG nodes, we need additional device-specific processing in software to interpret the results provided by the NIC. This

interpretation is provided by the device dependent node `IPv4Csum` in our example PRG in Fig. 3.5.

These device dependent nodes are assumed to be developed with the NIC PRG and are expected to be provided in the form of device driver library (shown as `DDLib` in the *Device dependent layer* in Fig. 4.2). It is also assumed that these functions can be executed independently within an RX-TX queue-pair as execution engines deployed in different protocol threads can invoke these functions independently on their queue-pairs.

Implementation of special nodes: As part of the partitioning phase (described in section 4.2.1), Dragonet introduces special nodes to enable communication between different partitions. These special nodes are responsible for enqueuing and dequeuing packet buffers in the communication channel and, if multiple edges cross the partition, multiplexing and demultiplexing them over the communication channel.

The implementation of these special nodes depends on the communication channel used and may need additional information from the execution engine during runtime. The Dragonet control thread is responsible for providing the necessary information to the execution engine to enable correct execution of these special nodes.

Next, we describe the implementation of shared memory based bulk transport channels for providing communication between different execution engines and the packet transfer.

Buffer management

The Dragonet network stack has the buffer allocation and free operations in its critical path as any f-node can allocate or free the buffer. Also, these packet buffers need to travel between different partitions which might be running in a different address space and trust domains. Therefore, we needed a low overhead and cross process buffer management and communication system.

The communication among separate partitions of the graph is implemented by using a bulk transport library, implemented as a part of the Barrelfish project [JB14]. This library provides unidirectional shared memory single producer, single consumer bulk transport library and is based on URPC [BALL91] and Barrelfish UMP communication channels [BPR+11].

This communication channel works by registering a dedicated memory pool, which is used for allocation of fixed size buffers. The dedicated pool and fixed size buffers reduce the overheads of buffer management.

This buffer management library allows passing a buffer with and without ownership hence providing security primitives required by the Dragonet data path.

4.3.2 Control Plane

The main objective for the control plane is to generate and maintain efficient data plane implementations by using the available information.

We have already discussed how Dragonet generates partitioned LPG as an execution plan based on the current network state, available hardware features and resource allocation policies. In this section, we will focus on how the Dragonet control plane use partitioned LPG to generate and maintain the data plane.

The Dragonet control plane is implemented by a single control thread which oversees the whole execution of the network stack (labeled as *Dnet Control thread* in Fig. 4.2). This thread manages the execution engines deployed in the protocol threads and in the applications using the **Data plane management** mechanisms described below.

Data plane management interface

The data plane management interface allows the Dragonet control thread to manage partitioned protocol graphs deployed in the execution engines. These execution engines can be either in a Dragonet protocol thread or in

the application context. In addition, this interface allows direct manipulation of the task queue of these execution engines and the configurations of the communication channels between different partitions. These interfaces are shown with dotted black line labeled as *Data Plane Mng.* in Fig. 4.2.

The Dragonet control thread uses the Unix socket channel created for the application endpoint to communicate the control commands with the application. The communication with the protocol threads is easier as they are implemented in the same process.

The Dragonet control thread can use the data plane management interface to start and stop the execution engines, and manipulate their task queues by spawning new tasks directly.

This interface also provides the primitives for creating a node, adding a port to a node, adding an edge and clearing a graph. These graph operations can be used to manage partitioned protocol graphs used by the execution engines in the protocol threads and the applications.

As the Dragonet control thread needs to manage the execution engines and the partitioned protocol graphs running in the applications, we provide a library which each application needs to use. This library provides communication with the Dragonet control thread and implements the *Data plane control interface*, allowing the Dragonet control threads manage the execution engine in the application. This library is shown with the label `DNetAppCtl` in the Fig. 4.2).

The Dragonet control thread manages the use of the communication channels by providing the appropriate parameters to the special communication nodes added to the partitioned protocol graphs.

Initialization of the network stack

The first responsibility of the Dragonet control thread is to initialize the device drivers and the basic network stack. The control thread uses the device driver library to initialize and configure the device driver in the initial state. The initial state of the network stack is assumed to have enough hardware resources to perform basic packet processing. For example, the initial NIC

configuration should direct all incoming packets to the default RX queue, and the initial LPG should implement basic packet processing, including handling of ICMP and ARP packets.

The Dragonet control thread is also responsible for providing a communication endpoint for all applications which want to use the Dragonet network stack services. As the control path is not in the performance critical path, we have implemented this communication using the Unix domain sockets [SFR04]. Applications can connect to the Dragonet control thread over the Unix domain socket provided by the control thread, and send their requests using the *network flow interface*. This Unix domain socket based communication channel is also used by the application to implement the *Data plane control interface*. The Dragonet control thread can use it to send the control command commands in order to maintain the execution engine in the application address space.

Maintaining the network state

Once the device driver initialization and communication endpoint creation is done, then the Dragonet control thread continuously maintains the network state by performing following steps:

1. It accepts application requests for network flow manipulations. These requests arrive from the *network flow interface* (3.5.2).
2. It executes the search to find a good hardware configuration (3.6.2).
3. It uses the hardware configuration and the current network state in the form of LPG to create a partitioned embedded graph which is used to generate specialized network stack deployment plan (4.2).
4. The control thread then updates the data plane (4.3.1) across the protocol threads and applications using the *Data plane management interface* (4.3.2) with the following steps:
    - Deploy the newly created partitions either by starting new protocol threads or by sending the partitioned protocol graph to an application for the execution.
    - Create new communication channels between the partitions if necessary.

- Update protocol graphs deployed in the protocol threads and applications.
- Update the NIC hardware configuration using the device driver library.
- Remove the unwanted communication channels.
- Stop the execution of unused partitions.

Currently the Dragonet control thread performs these operations sequentially, and there is further scope for improvement by performing few of these operations concurrently. As running above operations concurrently has correctness implications, it would involve careful thinking and analysis before adding such parallelism into the control thread.

4.3.3 Device driver layer

We have moved the NIC hardware dependent part of our runtime system into a device driver library which is used by both the control plane and the data plane to interact with the NIC hardware. The device driver library functionalities can be classified into data plane and control plane categories, and we have labeled these categories as `Q-mng` and `D-Control` respectively in the Fig. 4.2. If the library function is dealing with packet processing then we classify it in the data plane, and if the function is dealing with managing the configuration of the NIC then we classify it in the control plane category.

The interaction between the data plane and the device driver library happens from the PRG function nodes which were embedded into the LPG including the RX and TX queue nodes responsible for interacting with hardware queue endpoints. Their hardware dependent f-nodes can directly call the device driver library functions for packet handling.

The interaction between the control plane and the device driver library happens via the NIC configuration suggestions made by the hardware oracle (outlined in section 3.4.1). The Dragonet control thread uses the device control functionality of the device driver library (labeled as `D-Control` in Fig. 4.2) to set the selected configuration into the NIC hardware.

As the device driver library implements the configuration changes suggested by the NIC hardware oracle, these two components have to be compatible

regarding the supported configuration changes. We believe that this requirement is reasonable, as both of these components are expected to be developed together.

We have implemented device driver layers for Intel i82599 and Solarflare SFC9020 NICs. We have developed our layer by using the Dataplane Development Kit (DPDK) [int] for the i82599 NIC and the OpenOnload library [Sol10a] provided by the Solarflare for SFC9020 NIC. The development of this layer requires knowledge about multiple components in the Dragonet network stack. For example, the developer needs understanding of the hardware capabilities of the NIC, the way these capabilities are exposed by above libraries and it's implications, what type of configuration changes and interaction does Dragonet control and data thread require with the library, and how to safely implement these interactions without compromising the performance.

4.3.4 Application Interaction with Dragonet

The flexibility and adaptability of the Dragonet network stack also affects the way applications can interact with it.

There are three aspects to using the Dragonet network stack from the application perspective that we have covered so far. These include (i) submitting their requirements using the *network flow interface* (3.5.2), (ii) adapting to changes using the *Data plane management interface* (4.3.2), and (iii) data plane communications using the shared memory communication channels (4.3.1). Now, we describe the overall interaction of the application with the Dragonet runtime system.

Applications communicate with the Dragonet network stack by first connecting to the control thread over Unix domain sockets [SFR04]. Once connected, the application can register application endpoints with the Dragonet network stack. The creation of the application endpoint will include establishing a shared memory based communication channel between the application and the Dragonet network stack. This endpoint can be used for data plane communications. Once the application endpoint is created, the application thread then can use it to create new sockets and bind these sockets with the network flows using the *network flow interface*.

The application request is sent to the Dragonet control thread, which processes the request by updating the network state and use this updated state for the configuration space search, and generation of partitioned protocol graph to meet the application's request for the flow. Based on the partitioned protocol graph, the control thread determines what changes are needed in which components of the protocol graph, and sends these changes using the *Data plane management interface*. As application endpoints can receive these changes arbitrarily, they are expected to handle them in an asynchronous way. Handling these asynchronous events to manage the data plane execution engine in the application address space is another way applications interact with the Dragonet network stack.

Sending and receiving packets: Dragonet exposes low-level interfaces for sending and receiving packets to give full flexibility to the applications. Once a network flow is bound with a socket on the application endpoint, the application can use the endpoint to allocate a packet buffer, and send this packet buffer on the network flow by spawning a send task on the associated socket with the packet buffer. The packet sending task will be performed by the execution engine inside the application address space.

Dragonet provides an event handling interface which is used by the application threads to check for events on the given application endpoint. The event can be (i) an internal event implying a *Data plane management* command from the Dragonet control thread, (ii) the arrival of a new packet buffer on one of the sockets associated with the application endpoint, or (iii) no event implying that the application is free to do some other work. The arrival of a packet event provides information about which socket endpoint the packet has arrived for and a pointer to the buffers holding the packet contents and attributes. The application can free the packet buffer once it has finished processing it.

4.4 Putting it all together

Here, we will summarize all the components of the Dragonet network stack to clarify the whole system. The Dragonet network stack has three main

responsibilities: making resource management decision, specializing and deploying the network stack, and running the specialized stack. The first two responsibilities fall into the control plane and are handled by the control thread, whereas the last responsibility is part of the data plane and is collectively implemented by all the protocol threads.

The network stack can re-evaluate the current resource allocation plan when a new request comes either for control flow manipulation or explicitly requesting the re-evaluation of the resource allocation plan. The control thread updates and uses current network state encapsulated in the LPG (3.3.2) and the capabilities of the NIC modeled by PRG (3.3.4) to re-evaluate the current resource allocation plan. This re-evaluation process will start with the search of the NIC configuration space using the hardware oracle (3.4.1). The suggestions provided by the oracle are evaluated using the cost function (3.6.1) to find a single suggestion. This configuration suggestion will determine the new behavior of the NIC, and this information is then used to create new specialized network stack (4.2) optimized for the new NIC behavior and the current network state. The control thread then sends the updates to the protocol threads about deploying the newly created specialized network stack, and the protocol threads take over implementing the data plane (4.3.1) as using the task-based runtime system.

4.5 Conclusion

We have discussed how the information provided by the dataflow interface (C-PRG) can be used by the Dragonet runtime system to specialize the network stack for currently configured hardware capabilities and application requirements.

We have implemented the Dragonet runtime system to realize our ideas. The Dragonet runtime system can a generate custom control plane and data plane for a network stack based on the configured hardware capabilities. This runtime system can also adapt the control plane and the data plane based on the changes in the resource allocation plans.

In the next chapter, we will evaluate our design and the implementation in the context of concrete use cases of policy-based queue management.

Chapter 5

Queue management with Dragonet

5.1 Introduction

Modern NICs are equipped with a large number of hardware queue and filters. These hardware capabilities are useful resources to distribute the packet processing on multiple CPU cores and provide Quality of Service (QoS).

Much recent work addresses the problem of maximizing raw performance in host networking stacks [JWJ+14, HMCR12, PSZM12, JJ09] by using the hardware queues and filters to ensure that each packet is processed on a single core, distributing data structures to avoid contention and batching. The availability of dedicated hardware queues and hardware filters can also provide a way to eliminate OS overheads by using techniques such as user space networking [Sol10a]. Taking these techniques to their logical conclusion, the recent research proposes fully decoupling the data plane from the control plane at the OS level [PLZ+14, BPK+14]. All these approaches depend heavily on the use of NIC hardware queues to achieve isolation and high performance.

Efficiently allocating NIC queues across the network flows in a system is critical to performance and can enable higher degrees of service consolidation. In contemporary systems, however, there is no obvious, rigorous way to formulate, let alone solve, this problem: every NIC is different, and offers not merely different numbers of filters, but different semantics and limits for the filters and flow directors used to demultiplex incoming packets [SKRK13]. In addition to the diversity in the hardware, the allocation of these hardware resources also needs to adapt with changing workloads and changes in the policies used to guide these resource allocation.

Traditional network stacks either limit themselves to simple load balancing features like Receive Side Scaling (RSS) with a dedicated hardware queue for each core or leaving the queue allocation to either device driver with internal policies [PSZM12] or the administrator with external tools [eth].

We use the Dragonet network stack to address this challenge by allocating NIC queues to flows in a variety of different NIC implementations based on generic OS policies. We argue that the right place to implement this allocation decision is not in the NIC driver (as happens today), and certainly not in the NIC itself. Instead, NIC queue allocation should be performed in a hardware-independent way by the OS, and specifically by the network stack. Hence, Dragonet explicitly controls how NIC queues are allocated to network flows.

Our goal is not to provide a queue allocation solution to meet specific resource management policies and requirements. These policies and requirements can change significantly over time as application demands and NIC hardware change. Instead, we are interested in creating the proper abstractions to formulate the problem, and building a network stack that can systematically solve it.

In this chapter we demonstrate how our dataflow model helps us with capturing the capabilities of the NIC hardware for filtering the packets to the proper queue. Then, we show the value of the model in Dragonet, a network stack that selects NIC configurations that satisfy generic OS policies. Instead of hard-coding policies, we express them using cost functions. Using the Physical Resource Graph (PRG) abstraction and information about the network state, Dragonet explores the NIC's configuration space for a configuration that minimizes the cost function. Next, We evaluate Dragonet in

section 5.6 using a UDP echo server and memcached on two modern high performance NICs (the Intel i82599 and the Solarflare SFC9020), and two policies: load balancing and performance isolation for a set of given network flows. Finally, we show that proper NIC queues allocation enables performance isolation and predictable performance in a portable way. Furthermore, we show that Dragonet finds good NIC configuration solutions with reasonable overhead to the control plane operations.

We begin with a discussion of our motivation and background for this work.

5.2 Motivation and Background

The work in this chapter is motivated by combined trends in processors (and associated system software) and networking hardware.

5.2.1 Network hardware

We make two arguments. First, exploiting NIC hardware queues is essential for keeping up with increasing network speeds in the host network stack. Second, doing so requires dealing with complex and hardware-specific NIC configuration.

Speed of networking hardware continues to increase; 40Gb/s adapters (both Ethernet and Infiniband) are becoming commoditized in the datacenter market, and 100Gb/s Ethernet adapters are available. The data rates that computers (at least when viewed as components of a datacenter) are expected to source and sink are growing.

At the same time, the speed of individual cores is not increasing, due to physical limits on the clock frequency, supply voltage, and heat dissipation. As with other areas of data processing, the only solution to handle higher network data rates in end systems is via parallelism across cores, which requires multiplexing and demultiplexing flows in hardware before they reach software running on general-purpose cores.

Fortunately, all modern NICs support multiple send and receive queues, and include filtering hardware which can demultiplex incoming packets to different queues, typically ensuring that all packets of the same "flow" (suitably defined) end up in the same serial queue.

Multiple send and receive queues in NICs predate the multicore era: their original purpose was to reduce CPU load by offloading packet demultiplexing. This also had the useful property of providing a mechanism for performance isolation between flows without expensive support from the CPU scheduler. However, modern NIC functionality is highly sophisticated and varies considerably across different vendors and price points.

In this chapter, we only focus on the receive path, where the hardware filtering and multiple queues play a crucial role of classifying and directing received packets to the appropriate receive queue. Note that the claim here is not that the transmit case is irrelevant. Instead, managing receive queues as a system resource is a sufficiently complex problem in itself, and hence, we are limiting our focus on it.

5.2.2 Diversity and Configurability of filters

The primary challenge in exploiting NIC queues is the diversity of NIC filters and the configuration complexity. NICs offer a rich and diverse set of programmable filters for steering packets to hardware queues. For example, the Broadcom NetXtreme II Family NICs [Bro08] support a small number of generic filters using bitmask matching, while Intel NICs [Int10b, Int10a] support a larger number of filters which are less flexible but simpler to configure. In addition, a single NIC can support different types of filters. Here we revisit the discussion about the Intel i82599 NIC [Int10b] from earlier chapter (3.3.3) to further discuss the behavior of the filters.

The Intel i82599 NIC exports 128 send and 128 receive queues and supports:

1. *5-tuple filters*: 128 filters that match packets based on five fields: `<protocol, source IP, destination IP, source port, destination port>`, each of which can be masked so that it is not considered in packet matching.

2. *Flow director filters*: These are similar to 5-tuple filters, but offer increased flexibility at the cost of additional memory and latency (they are stored in the receive-side buffer space and implemented as a hash table with linked list chains). Flow director filters can operate in two modes: "perfect match", which supports 8192 filters and matches on fields, and "signature", which supports 32768 filters and the matches on a hash-based signature of the fields. Flow-director filters support global fine-grained masking, enabling range matching. These filters also support priorities to handle the situation where more than one filter matches.
3. *Ethertype filters*: these filters match packets based on the Ethertype field (although they are not to be used for IP packets) and can be used for protocols such as Fibre Channel over Ethernet (FCoE).
4. a *SYN filter* for directing TCP SYN packets to a given queue. This filter can be used, for example, to handle denial-of-service (DoS) attacks.
5. *FCoE redirection filters* for steering Fibre Channel over Ethernet packets based on FC protocol fields.
6. *MAC address filters* for steering packets into queue pools, typically assigned to virtual machines.

Finally, the Intel i82599 also supports Receive Side Scaling (RSS) [HdB13, Mic] to spread the incoming traffic on multiple cores by using multiple receive queue. It works by generating a hash value from the incoming packet fields and using this hash value to index a 128-entry table with 4-bit values indicating the destination queue. We will revisit the implications of using RSS in next section.

In contrast to the Intel i82599 NIC, the Solarflare SFC9020 [Sol10b] NIC supports 1024 send and 1024 receive queues, 512 filters based on MAC destination and two kinds of 8192 filters each for TCP and UDP: a full matching in which the entire 5-tuple is considered, and a wildcard mode in which only destination IP and port are used. If a packet is matched by multiple filters, the more specific filter is selected. Moreover, each filter can use RSS to distribute packets across multiple queues (two different hash functions are supported).

In our exploration, we have targeted the Flow director filters and 5-tuple filters in the Intel i82599, whereas we use the 5-tuple filters in both full

matching and wildcard mode for the Solarflare SFC9020 [Sol10b] NIC.

5.2.3 System software

We now examine the evolution of network stacks and make two observations: modern network stacks indeed depend increasingly on NIC hardware to achieve good performance, but there is currently no solution that provides support for generic OS policies and deals with the complexities of modern NIC hardware.

RSS: Queue allocation in the NIC

Modern network stacks (and commodity OSes in general) have evolved incrementally from designs based on simple NICs feeding unicore CPUs. As multicore machines became dominant and the scalability of the software stack became a serious concern, significant efforts were made to adapt these designs to exploit multiple cores.

However, the difficulty of adapting such OSes while maintaining compatibility with existing hardware has limited the extent to which such stacks can evolve. This, in turn, has strongly influenced hardware trends.

For instance, the most common method for utilizing NIC receive queues is Receive-Side Scaling (RSS) [HdB13, Mic]. The main goal of RSS is to remove the contention point of a single DMA queue and allow the network stack to execute independently on each core. With RSS, the NIC distributes incoming packets to different queues so that they can be processed by separate cores. Packets are steered to queues based on a hash function applied to protocol fields (e.g., on a 4-tuple of IP addresses and TCP ports). Assuming the hash function distributes packets evenly among queues, the protocol processing load is balanced among cores.

The key drawback of RSS, as with any other hard-coding of policy into NIC hardware, is that the OS has little or no control over how queues are allocated to flows.

For example, RSS does not consider application locality. Maximizing network stack performance requires packet processing, including network protocol processing and application processing, to be fully performed on a single core. This increases cache locality, ensuring fast execution, and minimizes memory interconnect traffic, improving scalability. Hence, performant network stacks depend on a NIC configured to deliver packets to the queue handled by the core where the receiving application resides.

Queue allocation in the driver

The shortcomings of RSS can be addressed by using more flexible NIC filters, and trying to intelligently allocate queues to flows using a policy baked into the device driver.

An example of this approach is Application Targeted Receive (ATR)![Int13] (also called "Twenty-Policy" in [PSZM12]). This is used by the Linux driver for the Intel i82599, where, transparently to the rest of the OS, the device driver samples *transmitted* packets (at a rate of 1:20 by default) to determine the core on which the application sending packets for a particular flow resides. Based on the samples, the driver then configures the NIC hardware to steer received packets to a queue serviced by that core.

The high-level problem with driver-based approaches is that the NIC driver lacks a global system view (available network flows, current OS policy, etc.) to make good decisions. Instead of using the full information, it will use heuristics based on hard-coded policies that may create more problems than they actually solve.

Queue allocation in contemporary stacks

The queue allocation solutions for the specific requirements and policies have been added to the Linux network stack. For example, Receive Flow Steering (RFS) [HdB13] in the Linux kernel tries to address the poor locality of RSS and steer packets to cores on which the receiving application resides. When using RFS, the network stack keeps track of which core a particular flow was processed on (on calls to `recvmsg()` and `sendmsg()`), and

tries to steer packets to the queue assigned to that core. RFS without acceleration performs the steering in software, where Accelerated RFS uses NIC filters. In the latter case, drivers need to implement the `ndo_rx_flow_steer()` function, used by the stack to communicate the desired hardware queue for packets matching a particular flow. The driver is required to poll the stack for expired flows in order to remove stale filters. Currently, three NIC drivers (for Solarflare, Mellanox, and Cisco silicon) implement this function.

Another example is Affinity-Accept [PSZM12], which aims to improve locality for TCP connections. The incoming flows are partitioned into 4096 flow groups by hashing the low 12 bits of the source port, and each group is mapped to a different DMA queue, handled by a different core. The system periodically checks for imbalances and reprograms the NIC by remapping flow groups to different DMA queues (and hence cores).

Both of these methods are not without problems. Accelerated RFS operates on a very simplified view of NIC hardware. As a result, it cannot deal with the physical limits of the NIC (e.g., what happens when the NIC's filter table is full?), and at the same time cannot exploit all NIC hardware features. Affinity-Accept NIC queue management, on the other hand, targets a single NIC (the i82599), and cannot be applied to NICs that do not provide the ability to distribute flows based on the low 12 bits of the source port (e.g., Solarflare's SFC9020).

Perhaps more importantly, both of these techniques specifically target connection locality in a scenario in which all network flows are equal. It is not possible, for example, to utilize NIC queues to provide performance isolation to specific network flows.

Scalable network stacks

There has been some recent work [JWJ+14, HMCR12] in using NIC hardware queues and filtering capabilities to improve the scalability of a network stack. These efforts focused on improving the poor TCP performance for small messages and short-lived connections. Unsurprisingly, all of these works aim for good locality, i.e., ensuring that all processing for a particular network flow happens on the same core, which requires the use of NIC

hardware queues and flow steering filters to distribute packets among cores. MegaPipe [HMCR12] is a based on a redesigned API, and in addition to splitting up the acceptance of new connections among different cores, it batches multiple requests and their completion notifications in a single system call to improve performance. mTCP [JWJ+14] applies similar techniques to a user space network stack that interacts with applications via a traditional socket API.

These network stacks show the benefits of NIC hardware queues and filtering capabilities for scalability and application performance. However, they do not solve the problem of the diversity in NIC hardware capabilities.

Dataplane OSes

Recently, work in "dataplane OSes" such as Arrakis [PLZ+14] and IX [BPK+14] have proposed radically simplifying the design of the shared OS network stack. In particular, these systems attempt to remove the OS completely from the data path of packets. This is achieved using multiple queues, and adopting a hard allocation of queues to applications.

We believe that this structure will be increasingly compelling for high performance server OSes in the future. For this reason, we orient our work in this chapter more towards such dataplane-based OSes.

The adoption of dataplane-based designs, however, emphasizes the problem of intelligent queue allocation. For example, Arrakis [PLZ+14] specifies a hardware model for virtualized network I/O, called virtual network interface cards (VNICs). VNICs are expected to multiplex and demultiplex packets based on complex predicate expressions (filters). In contrast to traditional network stacks, the application is assumed to have direct access to the NIC DMA ring and establishing the proper filters (both on the send and the receive path) is not just a performance concern, but the mechanism for establishing protection across applications running on the system.

Real NICs, however, are not perfect: they have limited numbers of queues, limited numbers of filters, limited filtering expressiveness, and, as we mentioned in section 5.2.2, complex configuration spaces. Hence, in the context of a dataplane OS, the network stack is required to program NIC filters

based on application requirements and global policies (e.g., which applications should operate without direct hardware access due to limited NIC capabilities).

5.2.4 Discussion

Overall, we believe that the OS should be capable of dealing with network queues analogously to how it deals with cores and memory, since ignoring NIC queues can lead to problems. In an OS like Linux, for example, it is not possible to ensure performance isolation for applications that use the network without exclusively allocating one or more NIC queues to the application. In OS dataplanes, the problem becomes more extreme because protection (e.g., from applications spoofing headers) is achieved by exclusive queue assignment.

Furthermore, as services are consolidated, a single machine is expected to deal with complicated, diverse, and varying workloads, potentially served by multiple applications with different requirements. The OS, therefore, should be able to dynamically assign hardware resources such as cores, memory, and NIC hardware queues to applications to fulfil these requirements.

While well-known techniques exist for allocating cores and memory to applications, allocating NIC queues poses a significant challenge. For sending packets this is not a difficult task: the OS can just ensure that the queue is used exclusively by a single application. For receiving packets, however, allocating a queue requires ensuring that particular network flows are steered into a specific queue by the NIC. Different NICs offer different facilities for steering packets into queues, making queue allocation a non-trivial task. One way to perform this task, and how it is done in many cases in practice, is to manually select a static NIC configuration for a specific workload (e.g., via `ethtool` [eth] for Linux). This leads to reduced flexibility in deployment.

In this chapter, we argue that NICs should be configured by the core OS network stack based on the network state and given NIC-agnostic policies about how different network flows share resources. Moreover, this function-

ality should neither be hidden behind the device driver interface nor left up to manual configuration.

Dragonet, our prototype network stack that realizes our ideas, is driven by dataplane OSes as a primary use case, but we argue that NIC queue management is a problem common to both dataplane OSes and monolithic kernels, and the techniques we present are applicable to both.

5.3 Modeling HW Complexity

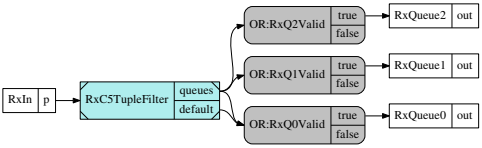
NIC filtering capability is an example of one of the complex NIC features. In this section, we show that the Dragonet configuration-node abstraction is sufficient to capture these complex hardware filters (5.3.1). We also discuss support from the c-node for incrementally configuring the filters, and the use of hardware oracles to reduce the configuration space of these filters (5.3.2).

5.3.1 Filter configuration space

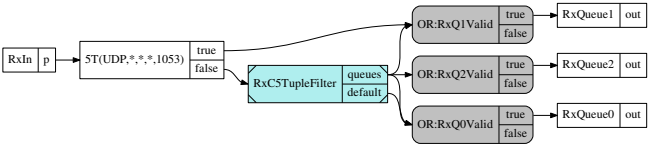
We have already discussed dataflow modeling (3.3) and how it can handle a general configuration space (3.4) in the previous chapter. Here, we use the configuration nodes (c-nodes) described in earlier chapter (3.3.4), and we show how we can use them to model the complex configuration space of the 5-tuple filters (described in 5.2.2) provided by Intel i82599 NIC as an example.

Our goal is to model the packet filtering capabilities provided by 128 5-tuple filters provided by the Intel i82599 NIC using the configuration nodes so that they can be incrementally configured and reasoned about.

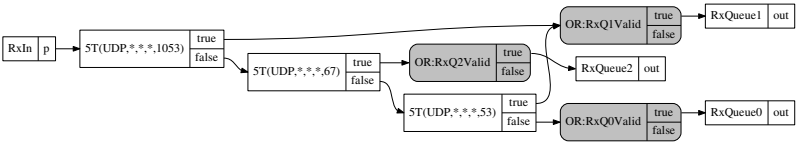
Fig. 5.1a shows an example of a receive side of a simplified version of the unconfigured PRG for Intel i82599 NIC with only 5-tuple filters. The function nodes (f-nodes) have a white background, operator nodes (o-nodes) have a gray background, and the configuration nodes (c-nodes) have a turquoise background. The `Q1`, `Q2`, `Q3` nodes represent the receive queues of the NIC.



(a) Unconfigured PRG



(b) Partially configured PRG after applying configuration `5T(UDP, *, *, *, 1053)` → `Q1`



(c) Configured PRG

Figure 5.1: Example of configuring 5-tuple filter in the Intel i82599 NIC

Instead of using a naive approach of adding 128 separate c-nodes, each modeling a single filter configuration, we have opted for simplifying the model by using a single c-node which can recursively replace itself with an f-node modeling a configured 5-tuple filter and a configuration node itself.

This recursive configuration can be seen in a partially configured PRG (Fig. 5.1b) in which an f-node `5T(UDP, *, *, *, 1053)` and the c-node with name `RxC5TupleFilter` have replaced a single instance of c-node with name `RxC5TupleFilter` from the U-PRG (Fig. 5.1a) after application of a single configuration `5T(UDP, *, *, *, 1053)` $\rightarrow$ `Q1`. The recursive expansion ends either after applying all available configurations or when all 128 filters are configured.

Fig. 5.1c shows the fully configured PRG after applying two remaining filter configurations on the `RxC5TupleFilter` c-node: [`5T(UDP, *, *, *, 67)` $\rightarrow$ `Q2`, `5T(UDP, *, *, *, 53)` $\rightarrow$ `Q1`].

In our example (Figures 5.1c and 5.1a), configuring `RxC5TupleFilter` adds three `5T` nodes. Hence, for all new edges the source node can either be the `RxIN` (since there is a `RxIN` $\rightarrow$ `RxC5TupleFilter` edge in the unconfigured graph) node, or a `5T` node. Similarly, the destination node can either be one of the `OR:Q`x nodes, or a `5T` node.

`5T` nodes represent an f-node implementing a configured 5-tuple filter of the i82599. A 5-tuple consists of the protocol, the source/destination IP address, and the source/destination port. In our example, the 5-tuple filters only specify the protocol and the destination port, leaving the other fields masked so that they can match all incoming packets for a given port and protocol, implementing a listen port. The example PRG models a NIC where UDP packets with destination ports `53` and `1053` are steered to `Q1`, UDP packets with destination port `67` are steered to `Q2` while all other packets end up in the default queue, `Q0`.

Dragonet uses boolean logic for reasoning. Each f-node port is annotated with a boolean predicate which describes the properties of the packets that will enable the port. Our expressions are built with atoms that are tuples of a label and value. The label typically corresponds to a packet field. For example, the predicate for the true (T) port of the filter node '`5T(UDP, *, *, *, 53)`' is: '`(EtherType, IPv4) AND (IPv4Prot, UDP)`

`AND (UDPDstPort, 53)`'. Note that it is not possible to have a different value for the same label. Hence, we can simplify expressions such as '`(UDPDstPort, 53) AND (UDPDstPort, 67)`' to false.

Just modeling the configuration space of these filters is not enough, and we need a way to select a few interesting configurations based on the current state of the network stack. We will discuss that next.

5.3.2 PRG oracle implementation

The number of hardware filters and the configuration space associated with these filters can be huge, making exhaustive exploration of this space unrealistic. We have presented the hardware oracle as a solution to this problem in the previous chapter (3.4.1), and here we show an example of hardware oracle in the context of queue management.

The hardware oracle provides a way to systematically reduce the configuration space by proposing only those configurations which will have an effect on the current network state, by using the NIC specific knowledge.

As we are focusing on the queue management, our oracle implementations for the Intel i82599 and Solarflare SFC9020 NICs generate configurations that map the given network flow to different queues by adding appropriate filters.

We give an example of potential suggestions generated by a simplified Intel i82599 NIC hardware oracle based on an input of a single UDP flow request and three hardware queues in Fig. 5.2. We used the flow `Flow (*:*, 127.0.0.1:1053, UDP)` as the input which requests for all UDP packets with `127.0.0.1` as destination IP, and `1053` as destination port, and no constraints on the source IP and the source port.

This example illustrates how the oracle can use the information about the flow to generate the most relevant configurations, hence reducing the overall search space significantly.

The hardware oracle is responsible for meeting the constraints on the hardware while generating these suggestions. It needs to ensure that the type of

```
    Flow(*:*, 127.0.0.1:1053, UDP) ->
  [
   5T(*:*, 127.0.0.1:1053, UDP, Q0),
   5T(*:*, 127.0.0.1:1053, UDP, Q1),
   5T(*:*, 127.0.0.1:1053, UDP, Q2),
   5T(*:*, *:1053, UDP, Q0),
   5T(*:*, *:1053, UDP, Q1),
   5T(*:*, *:1053, UDP, Q2)
  ]
```

Figure 5.2: Example of configurations suggested by hardware oracle for a single flow request.

filters to be used can actually support the suggested configuration. For example, the Intel i82599 NIC only supports global masking for *flow director filters*, which implies that all flow director filters have the restriction of having wild-cards on the same fields. Hence, the hardware oracle should either generate the first three (with two fields masked) or the last three suggestions (with three fields masked), but not a mix of them.

Similarly, the Solarflare SFC9020 NIC allows wild-card matching, but only on the source IP and source port fields, and hence the hardware oracle for SFC9020 NIC should not generate the last three suggestions in above Fig. 5.2.

The hardware oracle can also apply some basic NIC specific optimizations. For instance, the i82599 oracle will only use flow director filters if all the 5-tuple filters are used (see section 5.2.2).

5.4 Policy-based queue management

Our high-level goal is to offer policy-based queue management in a hardware-agnostic way. The Dragonet modeling approach provides a capability to reason about the NIC filters and queues in a portable way. However, we still need to figure out which are the important and active flows needing hardware resources. We also need to interpret the resource management policies provided to the system in the context of active flows and the current NIC configuration to perform policy-based resource management.

Here we discuss the approaches we have taken to understand the application requirements, and the resource allocation policies in a hardware-agnostic way.

5.4.1 Capturing application requirements: Active network flows

The first step for implementing policy based queue management is understanding which flows are active and important from an application's perspective.

Determining which flows are active is not trivial. Even for connection-oriented protocols like TCP, the fact that the connection exists does not mean that the connection is active (i.e., packet exchange might be minimal). Active network flows can be identified using traffic monitoring mechanisms, or registered directly by the applications.

We believe that the application developers are in the best position to decide which flows are active based on the application logic. Hence, we push this responsibility to the application developers by letting them specify what flows are active. We are using *Network flow* abstraction (described in the previous chapter 3.5.2) to gather application requirements for the network flows, and applications can use an additional attribute to mark the flows active. If more fine-grained metrics are needed (e.g., considering the traffic rate of each active network flow, rather than just whether it is active or not), our queue management algorithms can be easily adapted accordingly.

We have pushed the responsibility of specifying the active flows to the application, and we focus on selecting an appropriate NIC queue configuration for a given set of active network flows.

5.4.2 Implementing queue allocation policies

One approach for managing queues is to implement NIC- and policy-specific queue allocation algorithms. While it is possible to do so (easily) in Dragonet, such a scheme is problematic. Firstly, it requires that the policy imple-

menter understands the NIC operation and configuration details. Secondly, each allocation algorithm that implements a given policy would have to be rewritten for different NICs. Finally, such hard-coded allocation algorithms are difficult to compose. In a typical setup, multiple applications are running in the system, each contending for NIC queues, and each with a potentially different policy. In such a scenario individual application policies need to be combined with global policies.

A better solution is to decouple the policy specification from the details of NIC hardware. To do this we: (i) build NIC models that fully describe hardware operation and configuration, and (ii) describe queue allocation policies in NIC-agnostic manner. We have already discussed building NIC models (5.3), and here we focus on describing the queue allocation policies.

We express user policies via cost functions (5.4.4) that assign a cost to a specific queue allocation, given the set of active system flows in the system (5.4.1). Users can select an existing cost-function, or submit their own. Writing a cost function does not require any knowledge about the NIC. Furthermore, cost functions can be composed to form complex policies. A system-wide cost-function, for example, can split the cost into two parts: one representing the global queue allocation policy, and one representing the application policy. The latter can be determined by calling an application-specific cost-function.

Using a PRG and a cost function, Dragonet searches the NIC's configuration space for a configuration that minimizes the cost. As we discus in previous chapter (3.4) the configuration space is very large, rendering naive search strategies impractical. As a result, Dragonet applies several techniques to efficiently search the configuration space. We have already discussed the use of hardware oracle to reduce the configuration space (5.3.2). In addition, Dragonet uses qmaps (5.4.3) to simplify the execution of a cost function, and incremental search to support incremental changes in the state.

5.4.3 Mapping flows into queues

Ideally, the cost functions are expected to analyze the embedded LPG (discussed in the section 3.7.2 of previous chapter) graph which models how

LPG (representing required network processing) will be executed on top of configured PRG (representing the NIC configuration). But as the embedding operation is expensive, it is not ideal to perform it for every configuration suggested by hardware oracle.

We have taken a middle ground by making an assumption that all application requests are limited to flow manipulations and the hardware configuration modifications suggested by the hardware oracle are also related to the flow manipulations. We exploit these restrictions by generating a representation of *how flows are distributed in hardware queues* directly from the NIC hardware configuration. We call this flow distribution representation as *Qmap*. This optimization allows us to avoid performing the whole embedding for each potential NIC configuration suggested by the hardware oracle.

We can compute the flow mapping as follows. First, we apply the change to the configuration and use it to configure the PRG. Given a configured PRG, we can determine the queue on which a flow will appear using a flow predicate and a depth-first search starting from the source node (e.g., `RxIn` in Fig. 5.1c). For each of the flow's activated nodes, we compute the port that will be activated and continue traversing the graph.

In f-nodes, we determine the activated port by checking the satisfiability of the conjunctions formed by the flow predicate and the port predicates. We assume that the flow predicate contains enough information to determine which port will be activated (i.e., for each flow predicate, only one port predicate will be satisfiable). We had no issues with this assumption. Although boolean satisfiability is a NP-complete problem, in practice the flow and port expressions contain a small number of terms for this to become a restriction.

For o-nodes, we check the incoming edges and determine the activated port using the usual operator semantics. For example, in an OR node, the true (false) port is activated if the flow appears in the true (false) edge of *one* (*all*) operand(s). Note that for each operand, the flow can appear only on one edge (either true or false).

Computing the flow mapping dominates search execution time, and the method described above performs redundant computations. To improve search performance, we incrementally compute the flow mapping by main-

taining a *partially configured PRG* across search steps. Applying a *configuration value* to a c-node results in the c-node being removed. Applying a *configuration change* to a c-node maintains the c-node and results in a partially configured PRG.

The example of a configuration change is "insert a `5T(UDP, *, *, *, 1053)` $\rightarrow$ `Q1` filter". Applying this change to the graph of Fig. 5.1a results in the partially configured PRG of Fig. 5.1b.

To incrementally compute the flow mapping, we maintain information about how active flows are mapped in node ports in the partially configured graph. In Fig. 5.1b, for example, we can compute what flows match the T port of `5T(UDP, *, *, *, 1053)` (and will consequently reach `Q1`) and what flows match the F port. Note that c-nodes act as barriers, because we cannot compute flow mappings beyond them.

When an incremental change is applied, we propagate flow information to each newly inserted node. If the configuration change is a replacement, we recompute flow mappings for the affected nodes and propagate changes. As we show in our evaluation (5.6.5), incrementally computing the flow mapping leads to a significant performance improvement for the search algorithm.

5.4.4 Specifying policies with cost functions

Next, we discuss expressing queue allocation policies via cost functions that operate on a mapping of flows onto queues. We have discussed the role of the cost functions in evaluating the NIC configurations in the previous chapter (section 3.6.1), and here we focus on providing an example of it.

In a deployment of our system, we expect that some inbuilt policies will be provided to select from, as well as an interface that allows system administrators to give their own. We examine two policies as examples.

First, **load balancing** aims to balance the flows to the available queues. This policy is expressed easily using a cost function: we compute the variance of the number of flows in each queue.

Algorithm 1: Cost function for performance isolation policy

Input : The available queues Qs and flows F
Input : K queues assigned to HP flows
Input : A function `isHP()` to determine if a flow is HP
Input : The flow mapping $fmap$
Output : A cost value
`// determine HP and BE flows`
$(F_{HP}, F_{BE}) \leftarrow$ partition F with `isHP()` function
`// determine queues for each flow class`
$Qs_{HP} \leftarrow$ the first K queues from Qs
$Qs_{BE} \leftarrow$ the remaining queues after K are dropped from Qs
`// are flows assigned to the proper queues?`
$OK_{HP} \leftarrow \forall f \in F_{HP} : fmap[f] \in Qs_{HP}$
$OK_{BE} \leftarrow \forall f \in F_{BE} : fmap[f] \in Qs_{BE}$
if *(not* OK_{HP}*) or (not* OK_{BE}*)* **then**
  return *CostReject 1*
$B_{HP} \leftarrow$ compute balancing cost of F_{HP} on Qs_{HP}
$B_{BE} \leftarrow$ compute balancing cost of F_{BE} on Qs_{BE}
if F_{HP} *is empty* **then**
  return *CostAccept* B_{BE}
else if F_{BE} *is empty* **then**
  return *CostAccept* B_{HP}
else return *CostAccept* $B_{HP} + B_{BE}$

Second, we consider a policy that offers **performance isolation** for certain flows. We distinguish between two classes of flows: *high-priority* (*HP*) and *best-effort* (*BE*) flows. Following the data plane model, each class is served by an exclusive set of threads, each pinned on a system core, each operating on a single DMA NIC queue. To ensure a good performance to the HP flows, we allocate a fixed number of queues to be used only by these flows and leave the rest of the queues for the BE flows. As a secondary goal, each class provides its own cost function for how flows are to be distributed among the queues assigned to the class.

This example also illustrates the composability of cost functions, where each class may provide its own cost function (in this example we use load balancing) while a top-level cost function describes how flows are assigned to classes.

The cost function for this policy is also simple. Its implementation is 31 lines of Haskell code [1], and its pseudocode is presented in Alg. 1. It rejects all solutions that assign flows to queues of different classes, and returns an accepted solution with a score equal to the sum of the balancing cost for each class.

In our experience, cost functions, although in many cases small and conceptually simple, can be very tricky to get right in practice. The complexity in writing cost function comes from decisions like which configurations should be immediately accepted or rejected and how much penalty should be given to slightly undesirable configurations. Operating on the Dragonet models, however, considerably eased the development process because we could experiment and build tests for our cost functions without the need to execute the stack.

5.4.5 Searching the PRG configuration space

We have discussed how the policy-based resource management problem can be transformed into a search problem using cost functions to evaluate NIC hardware configurations in the previous chapter (section 3.6.2). Here we will show how we can search the PRG configuration space for a configuration that minimizes the cost function.

Search algorithm

We use a greedy search algorithm, which starts with a minimal configuration without any application-level flows and accepts a set of flows and a cost function as input. As the applications are expected to request a network service using the *network flow interface* (3.5.2), most LPG manipulations and resource allocation happens at the granularity of a flow. Hence, we use flow as a basic unit in our search for the configuration.

We opted for a greedy strategy due to its simplicity and because it can be applied incrementally as new flows arrive (see section 5.4.5). A simplified

[1] measured using David A. Wheeler's 'SLOCCount'

Algorithm 2: Search algorithm sketch

Input : The set of active flows F_{all}
Input : A cost function `cost`
Output : A configuration c

```
c ← C_0            // start with an empty configuration
F ← ∅              // flows already considered
foreach f in F_all do
    // Get a set of configuration changes
    // for f that incrementally change c
    CC_f ← oracleGetConfChanges(c, f)
    F ← F + f                                  // Add f to F
    find cc ∈ CC_f that minimizes cost(PRG, c + cc, F)  c ← c + cc
    // Apply change to configuration
```

version of our search is presented in Alg. 2, in which each step operates on a single flow (f) and refines the configuration from the previous step (c). At each step, we call the oracle to acquire a new set of configuration changes (CC_f) that incrementally modify the previous configuration. A configuration change can be applied to a configuration to form a new configuration ($cc + c$). We select the configuration change that minimizes the cost of the current set of flows (F), update the configuration and continue until there are no more flows to consider.

Depending on the problem, a greedy search strategy might not be able to reach a satisfactory solution. To deal with this issue, we allow cost functions to return whether the solution is acceptable or not in their cost value. An acceptable solution always has a lower cost than an unacceptable solution. If after the greedy search the algorithm is not able to reach an acceptable solution, algorithm "*jumps*" to a different location in the search space and starts again by rearranging the order of the flows. To minimize jumps, we support a heuristic where cost functions are paired with an additional function to sort the flows before the search algorithm is executed.

Incremental search

The above algorithm operates on all active flows. As flows come and go in the system, we need to consider that the optimal configuration might change. A naive approach to deal with added/removed flows would be to discard all state and redo the search. This approach, however, induces significant overhead and does not scale well as the number of flows increase. We can reduce this overhead by optimizing the search to work incrementally to deal with flow arrival and removal.

Adding flows is simple in the greedy search algorithm: we start from the current state and perform the necessary number of iterations to add the new flows. If an acceptable solution is not reached, we rearrange the flows (applying the sorting function if one is given) and redo the search.

Removing flows is more complicated to deal with. One approach would be to backtrack to a search state that does not include any removed flows, and incrementally add the remaining flows in the system. Because this can lead to large delays, we remove flows lazily instead.

As can be seen in Alg. 2, each flow is associated with a configuration change. When this change is applied to the PRG, a new set of nodes are added to the graph. When a flow exits the system, we maintain the configuration as is, and mark the configuration change that was paired with the removed flow as stale. This approach results in the nodes added by the configuration change to remain in the PRG, even though the corresponding flow was removed.

At some point, we need to remove the stale configuration changes. To do that we can backtrack the search as mentioned above. As an optimization, we allow oracles to repurpose the graph nodes that are associated with stale configuration changes when new configurations are needed. To support this, we define special configuration changes called *replacements*. In our current prototype, replacements are implemented by changing the predicates of the generated nodes for the replaced configuration change, but not the graph structure. The network stack can also periodically request to re-evaluate the queue allocation from scratch to flush all state configuration changes.

If the NIC does not have enough queues or filters to handle the all the flows

then the extra flows will be handled by the default queue, and then they will be demultiplexed by software to the proper application endpoint. The flows in default queue may get promoted to dedicated queue in future based on the availability of the resources when the network stack decides to re-evaluates the queue allocation plan from scratch.

5.5 Implementation

While it is possible to implement our queue management in an existing network stack (e.g., in Linux we can maintain all the needed information externally and use `ethtool` [eth] to update the hardware filters), we use Dragonet for our implementation because all the necessary abstractions already exist as first-class citizens, and Dragonet also allows us to customize the network stack.

Dragonet is written in Haskell and C: the Haskell code is responsible for implementing the logic while the C code implements low-level facilities such as communication with the NIC drivers and stack execution. We have already discussed the runtime of our stack in detail (4.3), and we summarize the relevant parts of the implementation here.

Dragonet runs in user space, and spawns a control thread and a number of protocol threads, each operating on a different receive/send queue pair. In each of these queue pairs, Dragonet connects a separate instance of the software stack implementation (i.e., the LPG). This approach ensures that all processing of a single packet happens on the same core. It also allows specializing the LPG implementation based on the properties of the NIC queue that it is attached to. For example, if a queue is configured so that no packets for a particular application endpoint are received, we remove the relevant nodes for steering packets in this endpoint from the LPG instance connected to that queue.

The LPG is transformed from Haskell to a C data structure and sent from the controller thread to the protocol threads. It is executed in the protocol threads by traversing the graph and calling C functions that correspond to f-node and o-node functionality. Our current prototype supports UDP, IP, and

ARP. All communication between Dragonet threads and application threads is done via shared memory queues.

The Dragonet driver for each NIC needs to implement: a PRG, its oracle, the shared memory queue for communication with Dragonet threads, and a function that accepts a PRG configuration and configures the NIC. We have implemented drivers for the Intel i82599 and the Solarflare SFC9020 NICs. The first uses Intel DPDK [int] while the second uses OpenOnload [Sol10a].

5.6 Evaluation

In this section, we evaluate our system. We investigate the performance benefits of using Dragonet smart queue allocation capabilities. We specifically examine the performance effect of enforcing performance isolation for specific client flows in a memcached server (5.6.3). Next, we quantify the search overhead for Dragonet to find an appropriate NIC configuration (5.6.5).

5.6.1 Setup

As a server, we use the Intel Ivy Bridge machine with two sockets, and ten Intel Xeon E5 cores per socket (hyper threads disabled), and 264GB of total RAM. The server is running Linux (kernel version 3.13). The server is equipped with an Intel i82599 [Int10b] and a Solarflare SFC9020 [Sol10b] NIC.

For load generators (clients), we use different multicore x86 machines (with the same software as the server) using an Intel i82599 [Int10b] NIC to connect to the server over a 10GbE network. We always use the same allocation for client threads in the load generators to reduce the variance of the applied workload for each run.

Dragonet runs in its own process context (separate from applications) in user space using one thread per NIC queue (for polling NIC receive queues and application send queues), and one controller thread that runs the solver.

We allocate ten cores to the 11 Dragonet stack threads (and subsequently ten NIC queues), and the remaining ten cores to the server application. Although the protocol threads are not required to have an exclusive core, we do this because our queue implementation used for communication between the application and the protocol threads supports only polling and cannot block. This limitation of polling can be overcome by more engineering to either delivering the interrupt notifications to the protocol threads, or some other notification mechanisms which can reliably inform protocol threads about arrival of new packets in their hardware queue.

5.6.2 Basic performance comparison

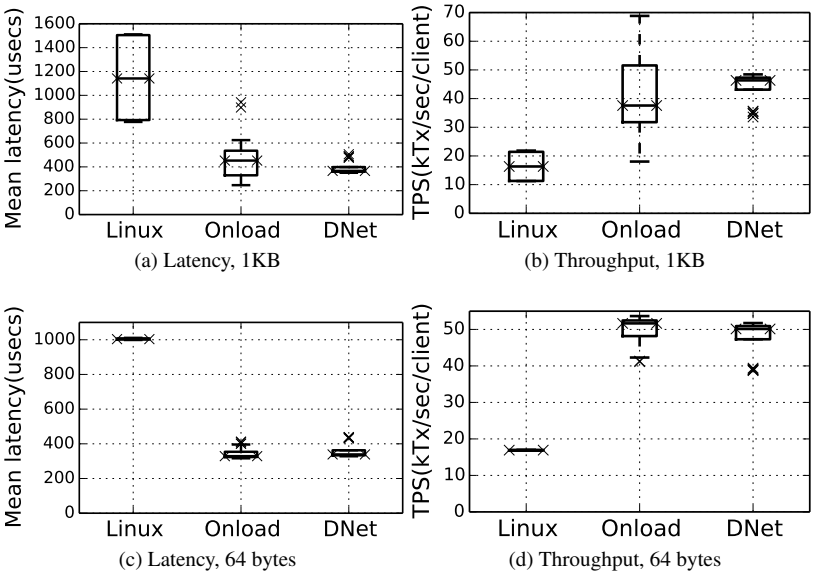
To put Dragonet's performance in perspective, we start with a comparison to other network stacks. We do not claim that Dragonet has the best performance. Our goal is to show that Dragonet has reasonable performance under the same conditions, and exclude the possibility that the benefits we report are artifacts of Dragonet's poor performance. To that effect, we are using simple resource management policy of load-balancing NIC resources in which flows are evenly distributed across queues policy for Dragonet and RSS for the other stacks.

We use a UDP echo server with ten threads, and generate load from 20 clients running on different cores on four machines. Each client runs a netperf [netc] echo client, configured to keep 16 packets in flight.

The results are presented in Fig. 5.3 for Solarflare SFC9020 NIC and Fig. 5.4 for the Intel i82599 NIC. We show boxplots for mean latency and throughput as reported for each of the 20 clients, using 64 and 1024 byte packets.

We compare Dragonet (*DNet*) against the Linux kernel stack (version 3.13) (*Linux*) and the high-performance Solarflare OpenOnload [SC08] network stack (*Onload*) using the Solarflare SFC9020 NIC, which we configure for low latency. We use Linux kernel stack as our baseline. The OpenOnload is a user-level network stack that completely bypasses the OS in the data path and can be transparently used by applications using the BSD sockets system calls.

The Linux network stack has the worst performance. For example, for 1024



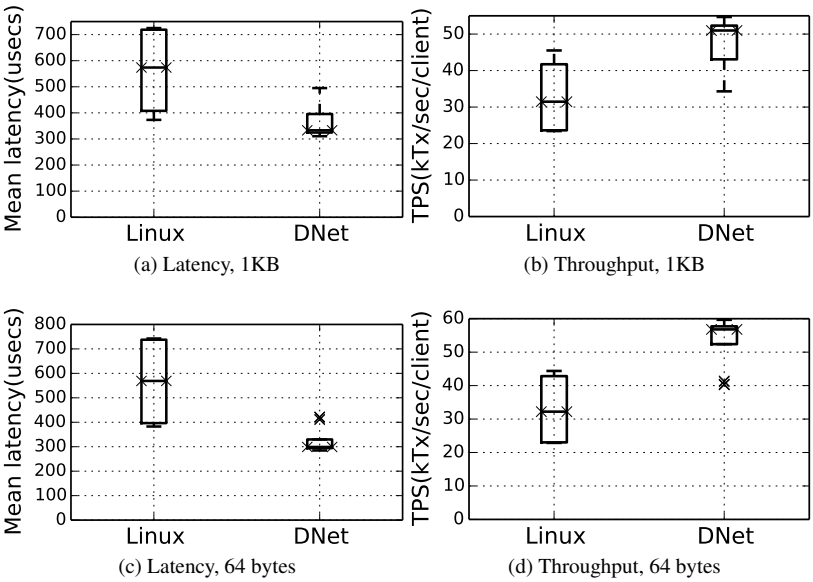
(a) Latency, 1KB (b) Throughput, 1KB

(c) Latency, 64 bytes (d) Throughput, 64 bytes

Figure 5.3: Comparison of echo server performance on the Solarflare SFC9020 NIC for different network stacks

bytes we measured a median latency of 1.14 ms and a median throughput of 16.3K transactions/s across clients. Also, the Linux network stack performance differs based on underlying NIC. For example, for 1024 bytes, clients observe 15K throughput on SFC9020, whereas they observe twice throughput (30K) on i82599. We believe that this is due different device drivers implementing optimizations like batching by second guessing application requirements.

Fig. 5.4 presents results of the Onload userspace network stack (*Onload*) on the SFC9020 NIC as a representative for high-performance network stacks. Unfortunately, we do not have an equivalent userspace solution for i82599 NIC. For Onload with 1024 bytes messages, we measured a median latency of 453 µs, and a median throughput of 36.6K transactions/s across all clients. The aggregate throughput of all 20 clients for Onload is 803K transactions/s, and the total transfer rate is 6.6 Gbit/s. These observations show that the



(a) Latency, 1KB (b) Throughput, 1KB

(c) Latency, 64 bytes (d) Throughput, 64 bytes

Figure 5.4: Comparison of echo server performance on the Intel i82599 NIC for different network stacks

Onload userspace network stack can achieve more than twice throughput by using the NIC hardware properly and optimizing the network stack for the NIC hardware.

For Dragonet on the SFC9020 NIC, we measured a median latency of 366 µs, and a median throughput of 46.3K transactions/s across all clients. The aggregate throughput of all 20 clients for Dragonet is 878K transactions/s, and the total transfer rate is 7.2 Gbit/s. Onload and Dragonet perform significantly better than Linux mainly due to bypassing the OS in the data path. Dragonet and Onload have similar performance. For 1024 byte requests, Dragonet outperforms Onload, while the reverse is true for 64-byte requests.

We also observed very similar throughput and latencies on the Intel i82599 NIC for Dragonet. This observation shows that, unlike the performance of the Linux network stack which can differ by 100% between these two

NICs, the Dragonet network stack can achieve consistent performance for both NICs.

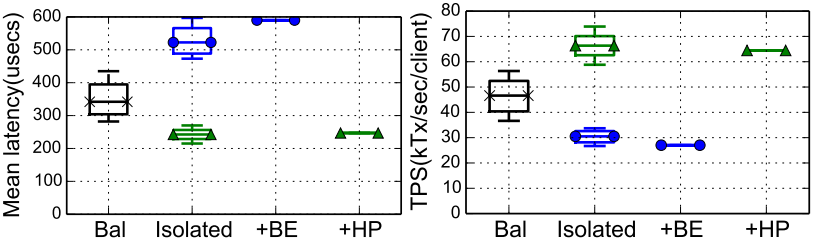
5.6.3 Performance isolation for memcached

In this section, we evaluate the benefits of smart NIC queue allocation using a (ported to Dragonet) UDP memcached server as an example of a real application. We consider a scenario in which a multi-threaded memcached [Dan13] serves multiple clients (e.g., web servers), and we want to prioritize requests from a subset of the clients that we consider high-priority (HP clients). We use the performance isolation cost function described in section 5.4.4 to allocate four out of ten NIC queues exclusively to HP clients. The memcached thread on queue-0 (which is the default queue) maintains a hash table to detect new flows and notify Dragonet of their presence.

Our experiment is as follows: we start a multi-threaded memcached server with ten threads exclusively using ten of the server's cores. We apply a stable load from two HP clients, and 18 best-effort (BE) clients, each with 16 flows, resulting in a total of 320 flows. We generate the load using memaslap [Dat13], a load generation and benchmark tool for memcached servers.

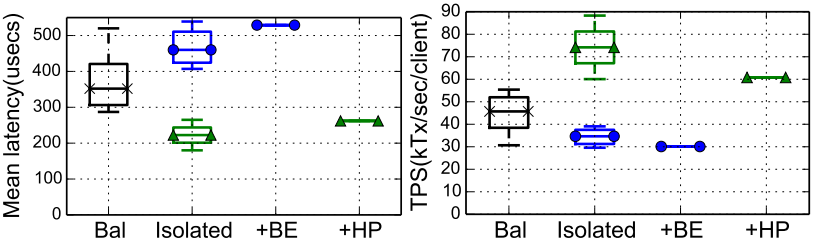
After 10 s we start a new BE client, which runs for 52 s. After the BE client is finished we wait for 10 s and start a new HP client, which also runs for 52 s. Each of the new clients is added as new flow, and the server triggers a search. We collect aggregate statistics from each client (mean latency and throughput), and show results for 1024 and 64 byte server responses for both NICs in Fig. 5.5 and Fig. 5.6 respectively. We use 10/90% Set/Get operation mix.

Each plot includes: (i) the performance of the workload under a load-balancing policy (*Bal*) for reference, (ii) the performance of the workload under the performance isolation policy (*Isolated*), (iii) the performance of the added BE client (+*BE*), and (iv) the performance of the added HP client (+*HP*). For the performance isolation policy, we use two different boxplots in our graphs: one that aggregates the HP clients (green color, median marked with triangles), and one that aggregates the BE clients (blue color, median



(a) Latency, Intel i82599 NIC

(b) Throughput, Intel i82599 NIC



(c) Latency, Solarflare SFC9020 NIC

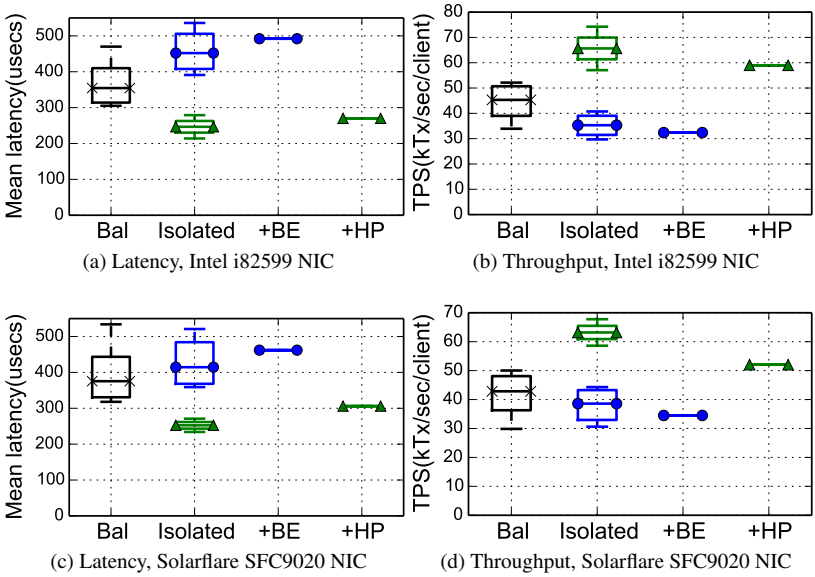
(d) Throughput, Solarflare SFC9020 NIC

Figure 5.5: Evaluation of memcached using 1024 byte messages with a priority cost function on Intel i82599 NIC and Solarflare SFC9020 NIC using ten queues.

marked with circles). For the load-balancing policy, we use one boxplot (black color, median marked with 'x') for all clients.

As an example, we consider the case of the Intel i82599 NIC for 1024 byte requests. Under a load-balancing policy, the median average latency across clients is 342 µs, the median throughput is 46.6K transactions/s, and the aggregate throughput is 927.5K transactions/s. Under the performance isolation policy, HP clients achieve a median latency of 246.5 µs (27% reduction compared to balancing) and a median throughput of 65.6K transactions/s (41% improvement compared to balancing). Furthermore, newly added HP flows and BE flows maintain the same level of performance as their corresponding classes in the stable workload.

For all cases, Dragonet queue allocation allows HP clients to maintain a significantly higher level of performance via a NIC configuration that is the



(a) Latency, Intel i82599 NIC

(b) Throughput, Intel i82599 NIC

(c) Latency, Solarflare SFC9020 NIC

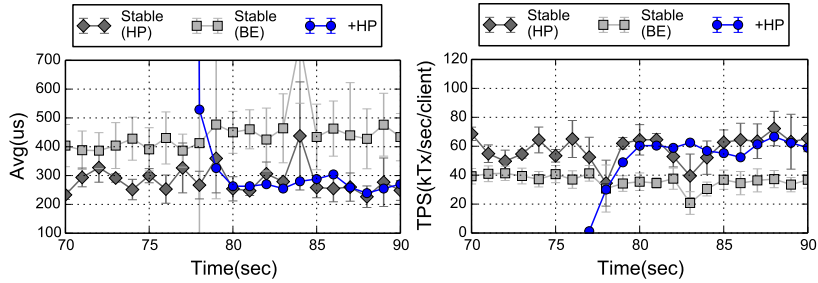
(d) Throughput, Solarflare SFC9020 NIC

Figure 5.6: Evaluation of memcached using 64 byte messages with a priority cost function on Intel i82599 NIC and Solarflare SFC9020 NIC using ten queues.

result of a NIC-agnostic policy. To the best of our knowledge, no other network stack enables this.

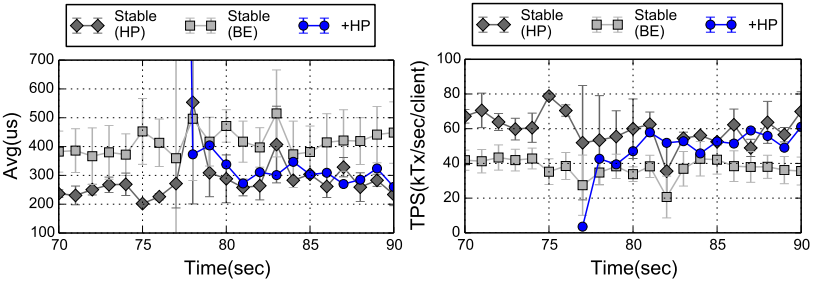
5.6.4 Impact of adding a dynamic flow

We also instruct memaslap clients to provide results for latency and throughput for each second (the minimum possible value). Fig. 5.7 shows our results for 64 byte requests, focusing on adding a HP client. It shows median throughput and latency for all clients in the initial workload, and the individual throughput and latency measurements for the new HP client. The initial latency of the HP client is high (12.6 ms for i82599 and 4.5 ms for SFC9020) and is omitted from the graphs for clarity. During the addition of the new client, performance of all clients drops for one second (sampling



(b) Latency (c) Throughput

(c) Intel i82599 NIC



(e) Latency (f) Throughput

(f) Solarflare SFC9020 NIC

Figure 5.7: Impact of adding a HP client when using 64-byte requests

period), but it quickly stabilizes again. We attribute these delays to the Dragonet solver executing and contending with the remaining Dragonet threads, and the time it takes to pass the new LPG graph to the protocol threads after a new configuration is found. We believe that careful engineering can eliminate the majority of these overheads. For example, in many cases, the LPG graph does not change across different configurations, so it is not necessary to reconstruct it in the protocol threads.

	Basic	Incremental flow map computation				
flows	full	full	+1 fl.	+10 fl.	-1 fl.	-10 fl.
10	11 ms	17 ms	2 ms	22 ms	9 µs	23.7 µs
100	1.2 s	0.6 s	9 ms	94 ms	74 µs	117 µs
250	13 s	4 s	21 ms	219 ms	190 µs	277 µs
500	76 s	17 s	43 ms	484 ms	382 µs	548 µs

Table 5.1: Search overhead for Intel i82599 PRG using ten queues

5.6.5 Search overhead

In this section, we examine the search overhead, i.e., how long does it take Dragonet to find a solution. For each possible new configuration given by the oracle, Dragonet computes how flows are mapped to queues (see section 5.4.3), which dominates the search cost.

Table 5.1 shows the search cost for a varying number of flows (ranging from 10 to 500) when using ten queues on the Intel i82599 PRG for the balancing cost function. The *Basic* column shows the cost of finding a solution without incrementally computing the flow mappings. All results in subsequent columns use incremental flow mappings computation. They show the cost of computing the solution from scratch (*full*), but also the cost of incrementally adding (*+1/+10 flows*) and removing flows (*-1/-10 flows*). For example, it takes 484 ms to find a solution for 10 new flows added when the system has 500 flows. Because we apply a lazy approach, removing flows has a small overhead.

The biggest challenge of our approach is reducing the search cost, which is not an easy problem. Our results show that incrementally computing flow mappings, not only allows to efficiently add and remove flows with small overhead, but also significantly improves the full computation because information is kept across search steps. The other approaches which use some properties of the flow to avoid using a solver for incorporating flows may work better, and we currently leave them for future exploration.

5.6.6 Discussion

Overall, our evaluation shows that Dragonet offers significant benefits by automatically configuring NIC queues. But, there is clearly a tradeoff: the search overhead. In general, the number of flows and the rate of changes in the workload determine the applicability of our approach. Considering two extremes, our system is well-suited for coarse-grained machine allocations in data centers for applications whose execution spans minutes, but cannot deal with load spikes in the order of a few milliseconds.

There are two aspects of the search overhead: constants and scalability in the number of flows. In this chapter, we focused on the latter and showed that incrementally computing the necessary information can significantly alleviate the overhead. We believe that there is significant room for improvement in both of these aspects. On one hand, we use a basic search algorithm that can be significantly improved. On the other hand, our profiling showed that more than 10% of the search execution time goes to basic operations (e.g., finding successors and predecessors) in the functional graph library [Erw01] we use. Moreover, more than 10% of the time goes to predicate computation done with our suboptimal library, even though we use Haskell's mutable hash tables [Col, LPJ94] to cache predicates.

We also note that while using Haskell for our prototype allowed us to rapidly develop our models and system, performance in many cases can suffer.

5.7 Conclusion

In this chapter, we argue for the increasing importance of exploiting NIC queues and filtering capabilities for scalable performance. We explore the diversity of NIC filters and the complexities of managing them, and we show a need for hardware agnostic approach to manage the queues and filters.

We present a policy-based queue management solution based on the Dragonet approach of using dataflow models to handle NIC complexity in a hardware-agnostic way. We show that the configuration node abstraction provided by the Dragonet can handle the complexities presented by the NIC

filters. We show that the cost function abstraction can be used to provide high-level policies in a portable way by implementing separate cost functions for load-balancing and performance isolation. We also present *Qmaps* and *incremental search* as the optimizations to allow Dragonet to manage a large number of flows.

We describe the implementation of Dragonet, which can use both Intel's *DataPlane Development Kit* and Solarflare's *OpenOnload* framework. We present experimental results using microbenchmarks and memcached for Dragonet's queue management, and we show the performance benefits of our approach by implementing and evaluating a load-balancing and performance isolation policy in a dynamic workload. We also quantify the overhead of configuration space search in our prototype implementation.

Our results show that the Dragonet approach of using dataflow modeling enables automated reasoning about the filtering capabilities of NICs. In addition, Dragonet can provide policy-based queue management by externalizing the resource allocation policies from the network stack in a portable way.

Chapter 6

Dragonet for Bandwidth management

6.1 Introduction

Interest in a host-level bandwidth management mechanism is increasing with the growing number of competing bandwidth-hungry applications. This is evident with recent work in data-center level network bandwidth management [JKM+13, HKM+13], and a research in implementing scalable rate-limiting capabilities using NIC hardware [RGJ+14]. Bandwidth management can provide better QoS and predictable performance by avoiding *cross-talk* between application flows using same underlying networking hardware. We believe that the ability to manage bandwidth between network flows based on high-level policies helps in making the network bandwidth a first-class citizen for effective resource management.

The current approach of providing bandwidth management capabilities in software, typically as a OS service, comes with an additional CPU overhead [RGJ+14]. This approach of providing bandwidth control as an OS service puts the OS in the data path, and hence is not a good fit with the recent push toward separating the data and control plane [PLZ+14, BPK+14].

Modern NICs are pushing the ability to provide bandwidth management in hardware which can save CPU cycles, but these features have limited capabilities and flexibility. For example, the Intel i82599 NIC supports upto 128 rate limiters, which are enforced on the transmit queues, instead of on the network flows. These capabilities are difficult to use due to their interactions with other NIC features, and we discuss them in section 6.4.1. In addition, the capabilities for bandwidth management differ across NIC vendors. This inflexibility and diversity in hardware capabilities makes it difficult to use them to implement bandwidth management in a portable way.

This chapter is motivated by potential benefits of providing bandwidth management and how NIC hardware capabilities can be useful for it. We aim to explore a hardware-agnostic approach to provide policy-based bandwidth management at host-level while using NIC hardware capabilities whenever possible. We present the Dragonet approach of modeling bandwidth management capabilities of NIC hardware, and bandwidth requirements of network flows as a data-flow graph. This abstract model enables reasoning about hardware capabilities for bandwidth management, and a current network state for bandwidth requirements. The high-level reasoning provides a way to implement a hardware-agnostic and policy-based bandwidth management on the abstract data-flow model.

We also present this work to showcase that the Dragonet approach is extensible beyond a queue management use case, and can be used to manage different categories of hardware features.

In the next section 6.2, we discuss a related work for different potential ways to provide network bandwidth management. We present a motivating use case of bandwidth management in deploying multiple analytics workloads in the following section 6.3. We make a case for using hardware capabilities by quantifying their benefits and difficulties in using them in section 6.4. In section 6.5, we present the Dragonet based solution and its evaluation. We discuss our conclusion in section 6.6.

6.2 Current approaches and related work

There are multiple ways to provide bandwidth management (e.g., software based, hardware based) and it can be provided at different levels (e.g., host level, network level). This section will review various approaches at different levels, while focusing on the flexibility of a bandwidth management approaches, and their ability to use hardware resources. We are mainly focusing on following aspects:

- **Bandwidth control enforcer**: This component responsible for implementing the bandwidth control using the available mechanisms. For example, bandwidth control can be implemented by a host software layer, host NIC hardware or a network switch.
- **Granularity**: This defines a degree of a fine-grained control supported for bandwidth management. For example, a flow-level granularity allows controlling the bandwidth of each individual flow, whereas a host-level granularity can only control the bandwidth for each host irrespective of number of flows from the host.
- **Bandwidth management interface**: This is the interface provided to specify the bandwidth allocation policies. This interface will be used by a resource management system or an administrator to manage the allocation of a bandwidth.
- **Flow requirements interface**: This interface specifies the active network flows and their desired network bandwidth requirements. The flow requirements are conceptually different from the actual bandwidth allocations as the former is the ideal allocation for the flow based on the application requirements, and the later is what a resource management system wants to allocate to the flow.

We describe current approaches available for network bandwidth management and traffic shaping using above aspects in this section.

6.2.1 Linux Traffic Control infrastructure

Linux provides a software based infrastructure (`TC`) [lin12] to implement traffic shaping functionalities. The `TC` bandwidth allocation policies work

by creating hierarchical traffic-classes using queue discipline (`qdisc`) definitions. The `TC` infrastructure provides `filters` to classify the network flows into the `qdisc` classes by using the network packet headers for matching.

The `TC` approach is flexible in creating detailed traffic classes and filters to suit traffic shaping requirements. This flexibility comes with a cost of high CPU overhead and limited accuracy in bandwidth limiting [RGJ+14].

The `TC` infrastructure focuses on enforcing traffic shaping, and leaves out the decision about how to allocate bandwidth between different flows and applications. It provides bandwidth management interfaces focused on creating traffic shaping classes and filtering the flows in these classes. The responsibility of understanding the flow requirements, making the bandwidth allocation plan and updating it based on the changes in a system is left to the higher-level service. Typically a system administrator is expected to make these decisions.

6.2.2 Software router based approach

Another approach is to implement a software switch within a host machine which is then configured to provide bandwidth management similar to a stand-alone network switch hardware.

Xen Open vSwitch [PPA+09] uses this approach to provide bandwidth control between different virtual machines. The Click router [KMC+00] is a configurable soft router which can be configured to provide bandwidth management functionality. A software router running within a host system can provide flexible bandwidth management at a granularity of networks flows, but with additional work for a CPU due to the software nature of the solution.

The soft router provides a rule-based interface, similar to a network switch configuration, focused on enforcing the bandwidth control. Similar to `TC`, this approach also leaves the responsibility of understanding the flow requirements and making the bandwidth allocation plan to other higher-level services.

6.2.3 Network based approach

A network-based approach provides bandwidth control as a function of the network by distributing it across all the switches in the network. It can be used by configuring every switch in a network to control the bandwidth used by each machine connected to it. This approach reduces the CPU overheads on the hosts by moving the functionality of bandwidth control to the network and distributing it across all the network switches. Unfortunately, this decentralization increases complexity of implementing policies due to the difficult problem involving correctly and consistently configuring of all the network switches.

The Software Defined Networking (SDN) approach separates the data-plane and control-plane of a network switch, while allowing full centralized programability of the control-planes of these network switches. This centralized control over control-planes streamlines network configuration [MAB+08], and hence simplifies the problem of managing the bandwidth controlling policies across all the network switches. The Openflow specification [spe12] provides flow-level bandwidth control mechanisms which are managed from a centralized SDN-controller to implement network-wide bandwidth control. The Openflow-based approach is successfully used for data-center scale bandwidth management for distributed applications in Google B4 [JKM+13] and Microsoft Swan [HKM+13] deployments.

Even with the increased flexibility of the SDN-based approach, using network switches for traffic control is not ideal in all the cases due to its reactive nature. These solutions allow excess traffic to be generated and then handle it by reactive actions such as random packet dropping [FJ93]. This reactive approach works well when the higher-level protocols have built-in flow-control mechanisms which can adapt to packet drops (e.g., TCP), but it may not work well for protocols which do not have adaptive flow control mechanism (e.g., UDP). Section 6.4 evaluates the interactions of external bandwidth control mechanisms on adaptive flow control mechanisms within protocols.

The use of `PAUSE` frames defined in the IEEE 802.3x [eth97] flow control scheme can avoid dropping packets by disabling further traffic generation for the whole link. This approach is too coarse grained when flow-level

bandwidth control is needed, as the PAUSE frames supports only link-level bandwidth control.

Data center bridging

Data center bridging (DCB) [dcb] aims to provide network-level traffic management solution using a support from both network and host.

DCB uses the Quality of Service (QoS) field in the IPv4 packet header to tag a packet with one of eight traffic classes. These traffic classes can be used to represent different requirements for the traffic (e.g. low latency, low jitter, high bandwidth, high priority, etc). DCB capable hardware and software elements in the network use these tags to classify the packets into an appropriate traffic class, and allocate the resources to them accordingly.

A network flow can be tagged with a traffic class based on the application suggestion provided as a flag in the socket API (SO_PRIORITY) [soc13], or based on the IPTables/TC rules [ipt] provided by a system administrator. Recent work uses an approach of inferring the traffic class for a network flow based on the net_prio cgroup of the application initiating the flow [neta].

Every DCB capable element in the network is free to interpret the meaning and resource requirements of these traffic classes independently, based on their local configuration. DCB tries to help with a potential mismatch due to a distributed configuration by supporting an information exchange protocol which can be used to share the configuration information across all DCB capable elements [dcb09].

In addition to tagging the traffic with a traffic class, DCB provides a mechanism for per-class bandwidth control using Enhanced Transmission Selection (ETS) [dcb11b], and can actively pause excessive traffic generation directly at a host using Priority based Flow Control (PFC) [Dcb11a] at a granularity of a traffic class instead of an entire machine.

The DCB approach provides some degree of separation of flow requirements from bandwidth allocation. The classification of flows into one of the eight

traffic classes is done by applications or a system administrator, and the actual bandwidth control can be implemented by any of the DCB capable elements in a network. This decentralization of bandwidth control provides an opportunity to move the CPU overhead associated with bandwidth control functionality to the most efficient location.

Even though the DCB approach can provide a network-level bandwidth management solution, it may not be suitable in all the cases as it is limited to only eight traffic classes. Also, it needs support from all networking elements and hosts in a network to provide effective bandwidth management, and hence limiting its applicability.

The DCB approach also introduces a global resource management problem which involves understanding the capabilities and constraints of all DCB capable elements and configuring them correctly in order to meet the network-level bandwidth management policies.

The standardization provided by the DCB approach has helped NIC vendors to push some parts of the bandwidth controlling functionality into NIC hardware. For example, the Intel i82599 NIC introduced partial support for PFC and ETS to offload part of the DCB functionalities onto NIC. We are exploiting these NIC hardware features in our work.

Our work is currently focused on solving the host-level resource management problem to meet the high-level policies while dealing with diversity and complexity in NIC hardware capabilities. Even though we are using the hardware capabilities in the NIC which were motivated by DCB, we are not tackling same problem of global resource management as DCB.

6.2.4 NIC hardware support for bandwidth control

Recent high end NICs are providing many hardware features including bandwidth control. One example is the Intel i82599 NIC which supports DCB features in the form of per queue rate control [Int10b] for up to 128 transmit queues and weighted scheduling of transmit queues to prioritize certain queues. There is very limited support for configuring these NIC hardware features in the mainstream Linux kernel [Bra12, Fas13b, Fas13a]. This limited support from the OS has led to the development of vendor specific so-

lutions such as the Data Plane Development Kit (DPDK) which exposes the hardware capabilities of NIC directly to applications [int]. The active developer community around this open-source project [dpd] suggests an increasing interest in exploiting the NIC hardware capabilities to implement high-performance networking applications.

The recent SENIC research explores the use of NetFPGA [Netb] hardware to implement precise bandwidth control in programmable hardware which can scale to tens of thousands of flows, without using a host CPU [RGJ+14]. This clean-slate approach of implementing a NIC using programmable hardware designed to provide scalable bandwidth-controlling capabilities makes a strong case favoring NIC hardware capabilities for bandwidth control.

We are currently focusing on exploiting the existing functionality of per queue rate control in the Intel i82599 NIC, but we believe that our approach of modeling the bandwidth controllers to provide policy-based bandwidth management can also be used with the SENIC design.

6.2.5 Software traffic shaping vs hardware bandwidth control

The bandwidth control support provided by a typical NIC should not be confused with the traffic shaping capabilities supported by a software solution like Linux `TC`.

A software based traffic shaping solution is typically flexible in supporting different characteristics of the network traffic. For example, Linux `TC` provides a Hierarchical Token Bucket (HTB) traffic shaper which supports separate configurations for traffic rate, burst size and bandwidth ceiling. In addition, it supports hierarchical composition of traffic classes, where each class can have a different traffic shaper, and a priority.

On other hand, typical NIC hardware implements a rate limiter which provides a limited configuration space. For example, the Intel i82599 NIC bandwidth controller will only ensure that the underlying transmit queue will send packets at a given rate. It neither supports separate configurations to accommodate different burstiness of a network flow, nor hierarchical composition of traffic classes or priorities.

In many use cases, the limited capabilities provided by hardware is sufficient to provide useful bandwidth control among the flows. In this chapter we are limiting ourselves to these use cases.

As we are focusing only on bandwidth control capabilities, we are using the terms "rate controller" and "bandwidth controller" interchangably in this chapter.

6.2.6 Putting Dragonet bandwidth management in context

The Dragonet approach for bandwidth management we present in this chapter tries to use NIC hardware capabilities whenever possible to reduce the CPU overhead. We are currently focusing on bandwidth management at the host-level using the hardware capabilities within the host, and we leave the problem of network-level bandwidth management for future exploration.

Dragonet also differs from the existing approaches as it separates the specification of network flow bandwidth requirements from the policies for actual bandwidth allocation. This allows us to develop and modify high-level bandwidth allocation policies without having to worry about the details of per flow bandwidth requirements. This separation of the policy also enables us to adapt the bandwidth allocation plan in presence of dynamically changing network flows and their requirements.

6.3 Motivating example: QoS for databases

This section presents an example showing some benefits of bandwidth management in providing Quality of Service (QoS) for a deployment of multiple analytics workloads on a single server. We aim to show that bandwidth management has an impact on observed client performance when bandwidth intensive workloads are competing with each other, and thus having the mechanisms to control bandwidth allocation is useful in such situations.

6.3.1 Use case overview

Our use case is inspired by a recent trend of running different types of data-processing on same data sources to provide different information [GMB+15], as well as co-locating different workloads on single machine. We are interested in the impact of sharing network bandwidth between different workloads, hence we choose to co-host multiple network intensive database analytics workloads to increase the contention for network bandwidth.

We show how to provide differentiated service to high priority (HP) workloads by provisioning more bandwidth than other best effort (BE) workloads. This use case is similar to the QoS use case we have seen in the queue management chapter (5.6.3), but here we focus on bandwidth management instead of queue allocation.

Application and deployment setup

We designed our experiment to quantify the impact of sharing the bandwidth between multiple applications and explicit bandwidth management. We used an in-house research database called Fruitbox [GMB+15], which is a distributed version of the SharedDB [GAK12] database. Fruitbox has two main components:

- **Storage engine**: This component is responsible for storing and accessing the database tables, and for basic operations like scans and lookups which are related to a storage layer.
- **Query engine**: This component is responsible for running all the operators (e.g., projection, join, aggregation) involved in executing a query.

The distributed version of the database allows running the storage engine and query engines on different machines, which communicate over the network using the `MVAPICH2` [Netd] library. This library provides an optimized implementation of `MPI` [Mes09] interfaces which can work over Infiniband, Ethernet and shared memory. For this experiment, we use a configuration which used the Linux socket interface on top of an Ethernet network.

The Fruitbox database supports running the Star-Schema benchmark [OOC07], a specialized version of TPC-H Analytics workload. The queries involved in the star-schema workload access most of the data, triggering a significant amount of data-movement from storage engine to query engine machines. This behavior makes it a bandwidth-bound workload, and hence is an interesting use case for explicit bandwidth management.

To ensure that the workload's bottleneck is the network bandwidth, we measured the bandwidth used by the system when running the workload, and confirmed that the it is indeed using most of the available bandwidth (8.8Gbps). The bottleneck was further confirmed by verifying that the reducing the available bandwidth to the machine leads to a decrease in application performance. Due to the read-only nature of the star-schema workload, most traffic is on a connection from storage engine to query engine, and hence we focus on these links in our evaluations and observations.

Hardware used

For our experiments we use two machines, each with four sockets and ten Intel Xeon E-5-4650 @ 2.4GHz CPUs on each socket, and 512GB total RAM. The machines are connected with the Intel 82599 10GbE NIC and FDR Infiniband interconnect. We dedicate one machine for storage engines and we use the other machine for query nodes and the client load generators.

Deployment details

In our experiments, we deploy four separate instances of distributed Fruitbox databases. All four storage engines are co-located on a single machine, and each storage engine is deployed on a dedicated NUMA node, to avoid performance degradation due to NUMA contention. Similarly all four query engines are co-located on a single machine.

We use core pinning and NUMA-aware memory allocation to reduce the contention for CPU and memory between different workloads. We use the Linux traffic shaping infrastructure to control the contention for network

bandwidth on the machine running the storage engines, and it is explained next.

Implementing differentiated bandwidth control

This section describes how we use the Linux traffic shaping infrastructure `TC` on the machine dedicated to the storage engines to control the bandwidth allocation between different instances of the database storage engines.

Based on our workload mix we create four classes for the network traffic so that each database instance has its own class. We mark one of the traffic classes as High Priority, and we provide a differentiated service to it by dedicating nearly half of the available bandwidth in a over-subscribed network link. We use the Hierarchical Token Bucket (HTB) bandwidth controller in our benchmarking, and we configure it to use the same bucket size, bandwidth ceiling and burst size to keep the configuration simple. Here is the full plan for bandwidth allocation:

- Root class: `9Gbps`, using Hierarchical Token Bucket (HTB) bandwidth controller
  1. Sub-class 1: Database-instance 1, `5Gbps`, high priority
  2. Sub-class 2: Database-instance 2, `2Gbps`
  3. Sub-class 3: Database-instance 3, `2Gbps`
  4. Sub-class 4: Database-instance 4, `2Gbps`

The Linux `TC` also supports borrowing of unused bandwidth between traffic classes. We create a separate "Static" and "Dynamic" allocation configuration using this feature. We also evaluate the "Default" behavior when no bandwidth management was applied.

6.3.2 Evaluations and observations

This section presents the results and observations from our experiments.

Figure 6.1 presents impact observed by the clients when running the four database instances using different bandwidth management configurations

for the storage engines. The plots include following bandwidth management mechanisms:

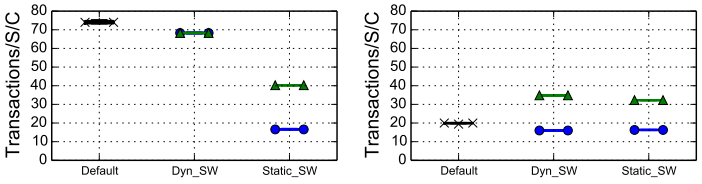
- **Default** setup: This setup uses the default behavior of no bandwidth management.
- **Dyn_SW** setup: This setup uses the software based Dynamic bandwidth management where applications are allowed to "borrow" unused bandwidth from other applications.
- **Static_SW** setup: This setup uses the software based static bandwidth shaping configuration where each application can use a fixed amount of bandwidth.

For each bandwidth management configuration we ran all four databases alone and in isolation from other workloads. This gave us an upper bound on how well a database can perform if there is no contention for bandwidth or other resources. Second, we ran all four database workloads together to measure the impact of contention for the network bandwidth.

In following plots, the high priority workload is represented as a green box-plot with triangles as a median mark, and the other three workloads in the best effort category are represented as a blue box-plots with circles as their median mark. The black box-plot with crosses as the median mark represents the workloads in the default configuration without any explicit bandwidth control.

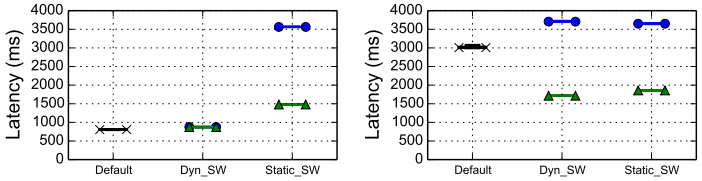
The box-plots for running the “Default” configuration (black box-plot with crosses as the median) *alone* and *together* shows that the transaction rate of each database drops to approximately `25%` when sharing the network bandwidth between four workloads. This is expected behavior as the default behavior of the Linux network stack is to ensure fairness for all applications including bandwidth allocation. Therefore each workload gets one-forth of the total bandwidth when running together, and hence the performance of each application degrades to one-forth.

The configuration “Dyn_SW” represents a bandwidth sharing policy where the goal is to maximize the utilization of the available bandwidth while providing differentiated QoS. This configuration is able to provide a high-priority database with approximately twice the transaction rate of the best-effort databases during bandwidth contention when running the workloads



(a) Running alone (b) Running together

Impact of bandwidth control Transaction rates



(c) Running alone (d) Running together

Impact of bandwidth control on Average Latencies

Figure 6.1: Impact of differentiated bandwidth control on a mixed analytics workload

together. It is also able to maximize the bandwidth utilization, and hence provide high transaction rates to all databases when they are running *alone*, without any bandwidth contention.

The configuration "Static_SW" represents bandwidth controlling policy where the goal is to provide stable performance irrespective of the contention on the bandwidth. The transaction rate for the high-priority database is approximately twice that of best-effort databases, irrespective of the changing contention when running *alone* and running *together*.

6.3.3 Observations about QoS in Databases

In this section, we used QoS for databases as motivating example, and we presented results supporting our claim that bandwidth management is an effective mechanism to provide differentiated Quality of Service between

bandwidth-intensive applications like analytics workloads. We also show that we need different policies to manage the bandwidth based on different high-level goals such as maximizing the link usage, performance stability or prioritizing certain workloads.

We assume that the configuration of network flows and bandwidth requirements from the above benchmark are representative of the QoS use case, and use them in the remaining chapter. Next, we will quantify the potential benefits from using hardware bandwidth controllers for the QoS use case used in this section.

6.4 Quantifying benefits of HW bandwidth controllers

In this section, we aim to quantify the benefits of using NIC hardware features for bandwidth management. Our main goal for the benchmarks presented in this section is to understand the impact of hardware rate controllers on CPU utilization and their accuracy, in comparison to software based solutions.

Our secondary goal is to understand the interaction of explicit bandwidth management on factors like packet sizes, and the flow control mechanisms present in protocols like TCP. At a high level, we make a case for benefits of hardware bandwidth controllers, and to explain the complexities in using these hardware features.

The next section 6.4.1 will give details of the hardware bandwidth controlling capabilities of the Intel i82599 NIC which we are using in this exploration. The following section 6.4.2 will explain the setup used in the benchmarking, section 6.4.3 will describe the results and our interpretation of them. The last section 6.4.4 will conclude the evaluation and will elaborate on the difficulties and implications using hardware bandwidth controllers on high level resource management.

6.4.1 Hardware bandwidth controllers in NIC

This section discusses the capabilities of bandwidth controllers in NIC hardware, and the tread-offs in using them.

i82599 NIC bandwidth management capabilities

As we discussed in the related work, most of the Intel i82599 NIC bandwidth management capabilities are motivated by Data Center Bridging [Int10b]. One example feature is multi-class priority arbitration and scheduling for bandwidth allocation between different virtual machines (VM) running on the same host machine using separate virtual functions provided by the NIC hardware per VM. Another useful feature is rate limiter (bandwidth controller) per TX queues to control transmit data rate for each TX queue separately.

Even though multi-class priority arbitration and scheduling is a useful mechanism for prioritizing a flow, it is tightly coupled with virtualization support and using it requires the use of separate NIC virtual functions (and hence separate MAC addresses). In our initial implementation, we focus on the rate limiting per TX queue feature, as it is easier to use, and is sufficient for our study of policy externalization.

This rate limiting is implemented by using the configured rate to calculate and enforce a minimum inter frame space between two consecutive packets sent on the same TX queue. This value is continuously adapted based on the length of the last transmitted packet and configured data rate to make sure that the configured data rate is enforced.

Advantages

Using the bandwidth controllers in NIC hardware can be useful to reduce the CPU load by decreasing the amount of work that needs to be done per packet by the CPU. Such a reduced CPU work per packet will become increasingly significant in the future as network speeds of 40 and 100 Gbps become more common.

Another potential benefit of using hardware bandwidth controllers is to avoid the need for a centralized bandwidth controller in software and hence avoid potential scalability issues. In addition, hardware bandwidth controllers can simplify software implementation for packet processing further by removing the need to buffer the outgoing packets in software for rate limiting. This allows a network stack to process each packet to its completion without any context switching, and hence makes the packet processing cache friendly.

In this evaluation we are focusing on the impact of the hardware bandwidth controllers on the CPU cycles consumed by the network stack, and we leave the evaluation of other potential benefits for future work.

Dis-advantages

Using hardware bandwidth controllers is a non-trivial problem as their applicability heavily depends NIC hardware capabilities, and the application requirements for bandwidth.

Additionally, bandwidth controllers tend to be tightly integrated with other hardware features like virtualization, DCB, and hardware queues which making them difficult to use without a detailed understanding of the NIC hardware. For example, In the Intel i82599 NIC [Int10b] the bandwidth controller can be either configured to work at the granularity of virtualized NIC interface or or at granularity of traffic classes in DCB, or at the granularity of a TX queue, but not at the granularity of a network flow. In addition, The bandwidth controller functionality differs between these modes To further complicate the matter, modifying the rate limiter settings for a queue without draining it beforehand can lead to undefined behavior. Using the hardware bandwidth controllers involves handling these types of quirks and understanding their implications for the rest of network processing.

6.4.2 Benchmarking setup

Motivation for experiment design: The goal for our benchmarking is to understand the impact of using hardware bandwidth controllers on CPU

utilization and on accuracy. For this purpose we select a streaming benchmark which stresses the CPU by doing large data transmissions. We use the stream benchmark from `netperf` [netc] in our benchmarks which has a similar behavior to the database QoS case-study we use in the previous section 6.3. We decided to use the `netperf` to generate the workload instead of database to simplify few engineering aspects. We don't aim to fully replicate the network trace of the database workloads, instead we plan to simulate similar network bandwidth contention to measure QoS impact.

We use both TCP and UDP streaming benchmarks in our setup to understand the interactions of protocol flow control with explicit bandwidth control. We set message sizes of 1024 bytes and 4000 bytes in our benchmark to understand the implications of packet sizes on the CPU utilization.

We have simulated the configuration of the database use case (section 6.3) by having four separate `netperf` instances connected to four separate `netserver` instances running on four client machines. We then mark one of the connections as "High Priority" and rest as "Best Effort" flows.

Machines details: In our experiments, the traffic generation is done by the Intel Ivy-Bridge machine with two sockets and each socket having ten Intel Xeon E5 cores and 256GB of total RAM. The server is running the `Ubuntu-14.04-LTS` Linux distribution (kernel version 3.13) and is connected to a switch using the Intel i82599 10GbeE NIC. As clients we use four 64-bit machines running same OS as the server which are connected to the same switch using the Intel i82599 10GbeE NICs. The clients in this benchmark are passive as they are only responsible for receiving the traffic, hence we do not focus on their configuration here.

Measurements: We measure the CPU utilization on the server by pinning a traffic generating thread to a specific core and then monitoring the load on that CPU using the monitoring utility `dstat` [dst]. In addition to the benchmarking tool `netperf`'s report of data sent and processed by each client, we also monitor the traffic received by each client to make sure that we do not count any traffic that did not reach the client applications.

Service Demand: In order to understand the impact of using hardware bandwidth management, we use the metric of "service demand". This metric describes the amount of time spent by the CPU per KB of network traffic. We calculate the service demand using observed CPU utilization on the participating server cores and the traffic observed by each client. The service demand metric gives a notion of work done by the server CPU in generating the traffic for particular flow.

We use the service demand metric as a way to compare the amount of work done by the CPU in generating the load when different bandwidth management mechanisms are used. Ideally, we expect to observe lower service demand when using hardware capabilities for the bandwidth management.

Implementing software based bandwidth management: We implemented software based bandwidth management by implementing a hierarchical packet scheduler using the Linux traffic shaping infrastructure `TC`, following the use case and traffic configuration from section 6.3.1 of co-located analytics workloads.

We created a separate traffic class for each `netperf` flow. We marked one connection as high priority and allocated `5Gbps` bandwidth, and we marked remaining connections as best effort and allocated `1Gbps` bandwidth for each of them. We used the Hierarchical Token Bucket (HTB) bandwidth controller configured with the same values for bucket size, burst size and ceiling for all our classes.

- Root class: `10Gbps`, using Hierarchical Token Bucket (HTB) bandwidth controller
  1. Sub-class 1: `netperf` connection 1, `5Gbps`, high priority
  2. Sub-class 2: `netperf` connection 2, `1Gbps`
  3. Sub-class 3: `netperf` connection 3, `1Gbps`
  4. Sub-class 4: `netperf` connection 4, `1Gbps`

We also used a `Dynamic` configuration which implements a work conserving bandwidth control by borrowing unused bandwidth between classes, and a `Static` and non-work conserving configuration which enforces the bandwidth limitations without allowing any bandwidth stealing at run-time.

Implementing hardware based bandwidth management: We used the "rate limiter per TX queues" hardware feature available in the Intel i82599 NIC and described in section 6.4.1. This hardware feature can restrict a transmit queue to a configured transmit rate, and we used this static rate control to provide basic bandwidth management.

Using these hardware bandwidth controllers in Linux is a non-trivial task as they are not supported by the Linux kernel [Bra12]. We modified the `ixgbe` device driver for the i82599 NIC to configure four specific TX-queues with specific rate-limiters with hard-coded values. Based on our configuration, we selected the values of `5Gbps` for the high priority queue and `1Gbps` each for the low priority queues. Only the traffic in these queues will be subjected to bandwidth management, and all other queues will work without any restrictions.

To control the network flows mapped to these customized TX queues, we use our insight into the network stack implementation. The Linux kernel network stack will typically allocate one TX queue for each core, and will use only that queue to send the outgoing traffic generated from that core. We exploit this behavior to select the TX queues with rate limiters by pinning the traffic generating thread to the specific core. This gives us some control on selecting TX queues in absence of explicit mechanisms to control the selection of the TX queues, and allowed us to run our benchmarks while using hardware bandwidth controllers.

Configurations used: We used the following four bandwidth controlling configurations in our benchmarks.

- **Default:** This configuration does not apply any bandwidth management. It is presented as a reference showing a default behavior of the Linux network stack.
- **Dyn_SW:** This configuration represents dynamic bandwidth control implemented in software using the Linux `TC` infrastructure. This configuration aims for maximizing the bandwidth utilization by allowing workloads to "borrow" unused bandwidth from other workloads while providing QoS when there is contention for bandwidth. This configuration is provided for a reference only as we are mainly interested in the evaluating the hardware capabilities.

- **Static_SW** This configuration represents static bandwidth control implemented in software using the Linux `TC` infrastructure. This configuration aims to provide the configured bandwidth irrespective of contention for the bandwidth.
- **Static_HW** This configuration setup represents static bandwidth control similar to Static_SW, but implemented using hardware features as described in a section 6.4.2. This configuration also aims to provide configured bandwidth irrespective of the contention.

For each of the above bandwidth controlling configurations, we run all four `netperf` workloads alone (in isolation from other workloads). This gives us an upper bound on data transfer and CPU utilization when there is no contention for network bandwidth. Second, for each of the configurations we run all four `netperf` workloads together. This allows measuring the impact of controlling the network bandwidth during the contention.

We select one of the workloads to be high priority (green box-plots with triangles as a median mark) and the other three workloads were grouped in the best effort category (blue box-plots with circle as a median mark).

6.4.3 Evaluations and observations

Figure 6.2 presents the data transmitted, CPU utilization and service demand for UDP with a message size of 1K and with different bandwidth management mechanisms.

Default bandwidth management:

The bandwidth observed by each client for all the configurations for running the workloads alone and running them together are presented in the sub-graphs 6.2a and 6.2b respectively. The "Default" configuration behaves as expected and is similar to the database case. The traffic observed by each client drops from `5Gbps` when running alone to `2.5Gbps` when running together. With four clients, essentially this benchmark is saturating the server NIC at `10Gbps` TX bandwidth.

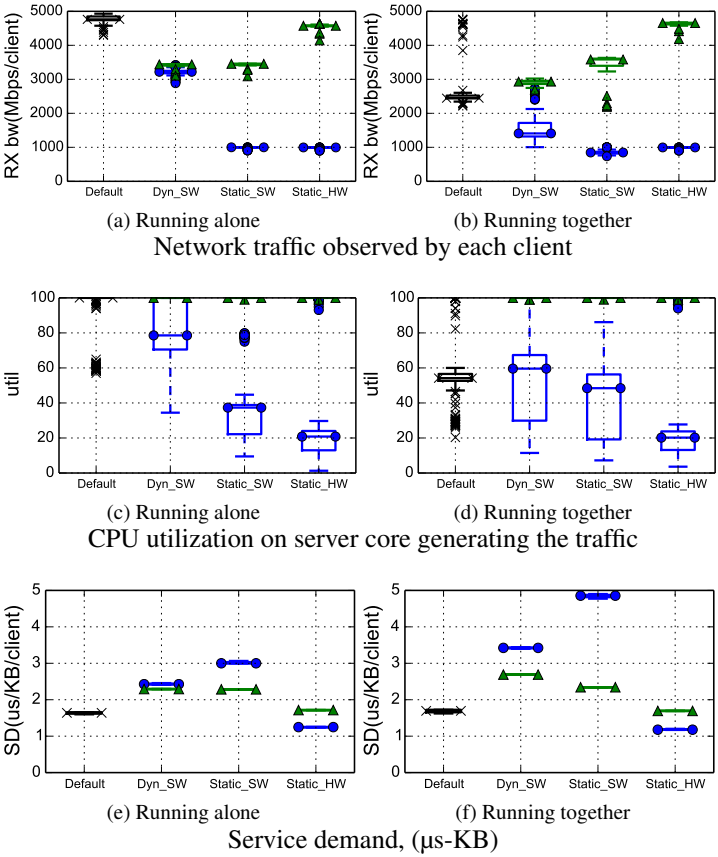


Figure 6.2: Understanding the impact of using different bandwidth management mechanisms with UDP for 1K message size

The sub-graphs 6.2c and 6.2d presents the CPU utilization including the "Default" configuration. The sub-graph 6.2c shows that when running alone this configuration saturates the CPU. In case of running all workloads together (sub-graph 6.2d), `35%` CPU is idle as the NIC TX capacity is saturated.

Note that 5Gbps is the maximum amount of traffic that can be generated with a single core when using a message size of a 1K as the single core becomes the bottleneck.

The most interesting observation for this configuration is that the service demand showing the time taken to handle 1KB of data does not change significantly between the workload running alone (sub-graph 6.2e) and running together (sub-graph 6.2f). This shows that amount of work done per data is constant in the default mode irrespective of other flows and load.

This meets our expectation that under the same bandwidth controller implementation, the service demand should be similar as no additional work is done by the CPU.

Dynamic bandwidth management:

The "Dyn_SW" configuration involves additional work on the CPU to maintain the per flow bandwidth rates in software. Due to this additional bookkeeping work, the service demand increases by 43% compared to the "Default" configuration even though there is no contention as each flow is running in isolation (sub-graph 6.2e). The additional CPU overhead leads to a reduced performance for the "Dyn_SW" configuration of less than 3.5Gbps bandwidth per client when each flow is running alone compared to the "Default" configuration which gets approximately 5Gbps bandwidth (sub-graph 6.2a).

The bandwidth borrowing works as expected and similar to the "Dyn_SW" configuration in the database workload (6.3.2), allowing low priority flows to borrow the bandwidth from high priority flows.

When running together the "Dyn_SW" configuration (sub-graph 6.2b) the best effort clients are able to get 1.5Gbps instead of the promised 1Gbps, and we believe that this is due to spare bandwidth available as the high priority flow is not able to utilize all of its 5Gbps bandwidth.

Static bandwidth management using software:

The "Static_SW" configuration behaves similar to "Dyn_SW" configuration regarding service demand when running the workloads alone as the network stack needs to do the additional work to enforce the bandwidth restrictions.

The service demand further increases for the best-effort workloads when running all workloads together for this configuration. The increase is particularly high for the best effort flows, and we think this increase in the CPU time per byte sent can be attributed to the extra effort needed to deal with additional buffering needed due to the limited bandwidth.

Static bandwidth management using hardware:

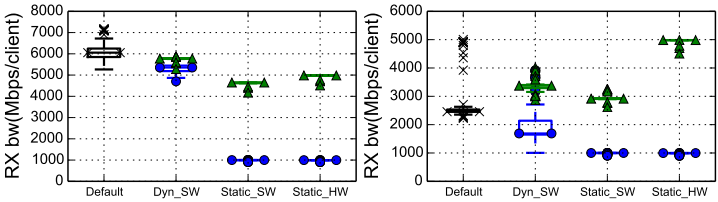
The service demand observed for the "Static_HW" configuration which uses the hardware bandwidth controllers is comparable to the "Default" configuration. This is the most interesting result showing that by using the hardware bandwidth controllers, the CPU does not need to do any additional work to enforce the bandwidth guaranties.

The interesting observation is that the actual bandwidth observed is closer to the configured value when using the hardware bandwidth controllers than for the software bandwidth controller. Also the CPU utilization is more stable when using the hardware bandwidth controller in comparison to the software bandwidth controller. The higher precision at lower CPU overhead provided by the hardware bandwidth controller in comparison to the software bandwidth controller is also reported by Radhakrishnan et al [Netb].

Evaluation with larger message sizes

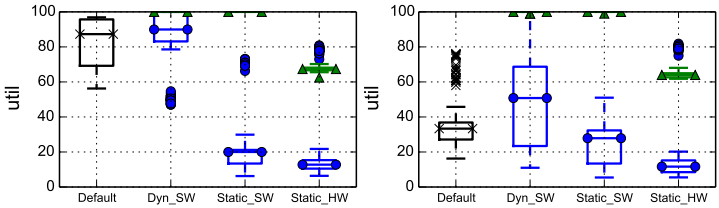
We also evaluated the impact of large message size which helps to reduce per-byte overhead by amortizing the packet processing.

Figure 6.3 presents the results for running the benchmark with messages of size 4K instead of 1K. These results show similar behavior as explained in



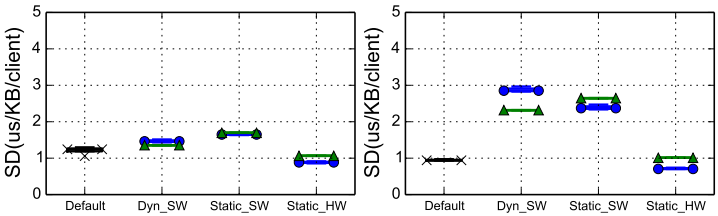
(a) Running alone (b) Running together

`dstat` RX Bandwidth at client, UDP, message size 4000



(c) Running alone (d) Running together

CPU utilization on server core, UDP, message size 4000



(e) Running alone (f) Running together

Service demand, (µs-KB), UDP, message size 4000

Figure 6.3: Understanding the impact of using different bandwidth management mechanisms on the CPU utilization with UDP for message size of 4000

previous experiment: the service demand is reduced when using the hardware bandwidth controllers.

Another observation is that the increased message size is adversely affecting the software bandwidth controllers in the presence of bandwidth contention, leading to an increased CPU utilization and reduced bandwidth for each

workload. Interestingly this degradation does not happen for the default configuration which does not implement bandwidth control.

We think that the synchronization overheads in maintaining the bandwidth statistics in the hierarchical data structures used by TC across the cores might be causing this performance degradation.

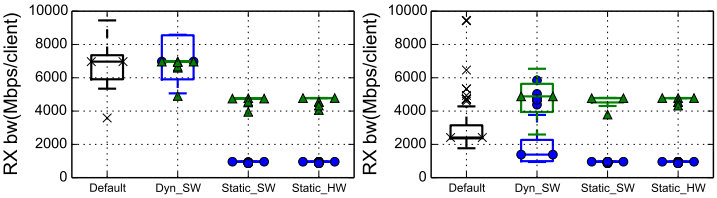
In contrast, the hardware bandwidth controller is getting more bandwidth while using much less CPU (around 40% less) and reduced service demand while meeting the bandwidth restrictions accurately. This supports the claim that hardware features will be more beneficial with higher transfer rates and larger bandwidth.

Evaluation with TCP

We evaluate the interactions of explicit bandwidth control and the flow control implemented by TCP by using our bandwidth controlling configurations with TCP. Our goal is to understand the implications of the interaction between flow control within a protocol and low level bandwidth control mechanisms.

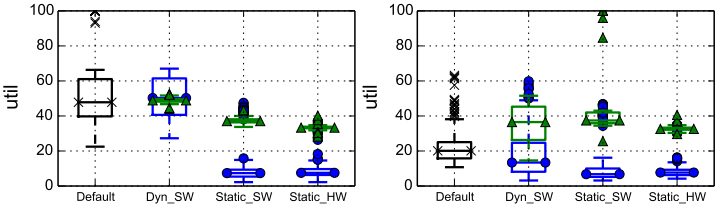
Figure 6.4 presents the results when running the benchmark using the TCP with 4K message size. The bandwidth observed by each client in this experiment shows that bandwidth control also works with TCP and can provide QoS between flows.

The important observation here is that the service demand does not change much with the different mechanisms of bandwidth management even though the bandwidth restrictions are obeyed correctly. We attribute the lack of change in the service demand to TCP's self correcting behavior where it adapts the window size based on the current availability of the bandwidth. This adaptive behavior leads to reduced CPU utilization as the CPU does not keep trying to send data all the time. Hence the benefits of reduced CPU utilization by using hardware bandwidth controllers are less visible for TCP in comparison to the stateless protocols like UDP which do not provide their own flow control. Using the hardware bandwidth controller can be also used to apply back-pressure at host-level to the TCP connection by controlling



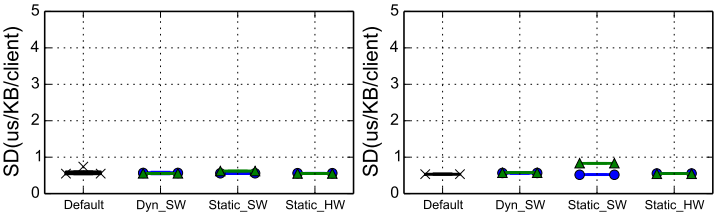
(a) Running alone (b) Running together

`dstat` RX Bandwidth at client, TCP, message size 4000



(c) Running alone (d) Running together

CPU utilization on server core, TCP, message size 4000



(e) Running alone (f) Running together

Service demand, (usec-KB), message size 4000

Figure 6.4: Understanding impact of TCP with different bandwidth management mechanisms on the observed bandwidth and CPU utilization

the transmission rate of the flow at the host level. This approach of providing host-level feedback to a TCP connection by generating back-pressure using hardware can be useful for data-center-wide bandwidth management techniques based on controlling the host transmission rates [GSG+15].

6.4.4 Observations about using HW capabilities for rate limiting

In this section, we used the rate limiting capabilities in the Intel i82599 NIC to show that bandwidth management hardware can save CPU cycles for protocols which don't have adaptive flow control (e.g. UDP). Even if these hardware capabilities may not yield CPU cycle saving for protocols with built-in flow control (e.g. TCP), they can be still useful as they correctly enforce the bandwidth limits based on the configuration.

Even though these NIC capabilities are useful, they are hard to use because of limited support from the OS to use these hardware capabilities. Using NIC hardware features also involves understanding their interaction with various other components of the network stack. For example, in our use of NIC hardware for bandwidth management, we had to understand and rely on the behavior of the Linux network stack to use the hardware TX queue associated with a local core to transmit packets from the processes running on that core. Similarly, you may have to worry about allocation of receive side hardware filters to make sure that incoming packets and their notifications are delivered to a receive queue which is linked to the core where the application is running.

The problem of using NIC hardware properly becomes even more difficult in the presence of multiple applications competing for the resources. Now it becomes necessary to worry about:

- deciding which flows should get which hardware resources
- making sure that other flows will correctly use software solutions
- configuring both the hardware and the network stack correctly
- adapting the bandwidth allocation plan when new flows arrive or old flows disappear

Without having a solution for all these parts, it is difficult to use NIC bandwidth management capabilities in the presence of multiple competing applications.

This difficulty is further increased because of different vendors implementing bandwidth management in different ways with different semantics. Dealing with this diversity in NIC hardware requires a portable solution for net-

work bandwidth management. Also, the policies for bandwidth allocation need to be portable and hence hardware agnostic, so they can be used irrespective of the hardware capabilities across different NICs.

We claim that policy-based usage of bandwidth management features provided by a NIC in the hardware-agnostic way can be achieved by applying the Dragonet approach of modeling the NIC hardware and network stack, and then solving a configuration space search and embedding problem. We present our solution in the next section.

6.5 Dragonet for managing HW bandwidth controllers

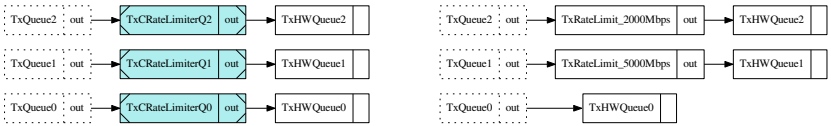
This section presents a Dragonet based solution for policy-based management of hardware bandwidth controllers to provide bandwidth control between flows.

Section 6.5.1 provides an overview of the solution and next section 6.5.2 presents an evaluation showing that the solution works as expected. Section 6.5.3 concludes the discussion.

6.5.1 Overview of Dragonet approach

This section provides an overview of how Dragonet provides policy-based management of NIC hardware bandwidth controllers in a hardware agnostic way. We have broken down the problem in the following sub-problems which we will explain in more detail:

1. Understanding hardware capabilities
2. Understanding application requirements
3. Generating potential suggestions
4. Evaluating potential suggestions based on high-level policies
5. Finding a good resource allocation plan
6. Implementing the selected resource allocation plan



(a) Unconfigured PRG (b) configured PRG

Figure 6.5: Example of bandwidth management configuration

Understanding Hardware capabilities: PRG

Dragonet needs to understand the capabilities and configurability of the NIC hardware regarding bandwidth management. This is done by modeling those hardware features and their configurability with a function node and a configuration node respectively in the NIC PRG.

Figure 6.5a shows the simplified example of a send side of an unconfigured PRG, which models the capability of the i82599 NIC to enforce per queue transmit rate limits for three TX queues. The function nodes are shown with a white background and the configuration nodes are shown with a cyan background. A function node with dotted boundary is a software node. The `TxHWQueue*` nodes represent hardware transmit queues of the NIC and the `TxQueue*` nodes represent software nodes which are responsible for handing over the send packet descriptors to the hardware for transmitting. The configuration node `TxCRateLimiter` represents the configurability of the NIC to provide per-queue rate limiting capability. This node can be configured with a number representing a bandwidth in `Mb/s`, or can be disabled to represent no rate control.

Figure 6.5b shows an example of a configured instance of the same PRG after applying the following configuration values [`TxRateLimit(5000Mb/s)` $\rightarrow$ `Q1`, `TxRateLimit(2000Mb/s)` $\rightarrow$ `Q2`, `TxRateLimit(None)` $\rightarrow$ `Q0`].

The `TxRateLimit_*Mbps` function nodes represent the configured hardware capability of the NIC to enforce specified bandwidth on the TX queue. In this example `TxHWQueue1` will be limited to the rate of `5000 Mb/s`

due to the F-node `TxRateLimit_5000Mbps`, `TxHWQueue2` will be limited to the rate of `2000 Mb/s` due to the F-node `TxRateLimit_2000Mbps` , and `TxHWQueue0` will have no rate limit.

The process of applying a configuration also adds a Boolean predicate specifying a rate limit (`TxRateLimit(5000)`) on the queue which will be used by Dragonet for the embedding step described later.

Understanding application requirements: Application Interface

In order to provide bandwidth management per flow, we need to collect information about the application's requirements for the desired bandwidth of each flow. As application developers are in the best position to know the desired bandwidths for the flows, we delegate this responsibility to them.

We extended the Dragonet application interface to allow applications to specify the desired bandwidth requirements for each flow. This information is optional and applications may choose to not provide it. In addition, providing this information does not guarantee the actual allocation of the bandwidth to a flow, but is used as a request to the system which may get ignored. For example, when the aggregated bandwidth requests from applications are more than the total available bandwidth, the system is free to ignore some bandwidth requests based on high level policies and application priorities.

Our interface is different from TC as we focus on gathering the bandwidth requirements for each flow from the applications, and high-level policies are used during the enforcement to decide which application requirements should be satisfied. On the other hand, the TC interface focuses on specifying how bandwidth should be divided and hence it combines the application requirements and the resource allocation policies. This focus makes the TC policies difficult to adapt when the application requirements change (e.g., one of the application finishes).

As the bandwidth requirement is an attribute of a network flow, we were able to model this attribute in our LPG by extending our existing flow abstraction 3.5.2. The bandwidth requirement attribute is later used by our

hardware oracle to generate reasonable suggestions for hardware configurations and by the embedding algorithm to determine if additional software nodes are needed to enforce these bandwidth requirements.

Generating potential suggestions: Hardware oracle

The next problem that Dragonet needs to solve is to efficiently explore the whole configuration space of the NIC hardware to find interesting configurations based on the current application requirements. The hardware configuration space of a NIC can be very large making it infeasible to search the whole space. The hardware oracle helps to reduce the search space by suggesting only a few reasonable configurations using information about application requirements, existing configuration and knowledge about the hardware capabilities.

To meet our goal of generating interesting hardware bandwidth controller configurations based on the application flow requirements, we extended our existing hardware oracle for the Intel i82599 NIC. This hardware oracle is already capable of suggesting hardware filter configurations for receive queues by analyzing the network flows, and we were able to incrementally modify it to include suggestions for rate controllers by looking at the desired flow requirements for bandwidth.

In order to simplify the problem of generating bandwidth management configurations, we make the assumption that "a network flow will use receive and transmit queues with the same queue-id". This allows the hardware oracle to allocate receive side hardware filters and transmit side hardware bandwidth controllers together.

Our existing Intel i82599 NIC hardware oracle use the flow information to propose a receive hardware filter to steer the flow on one of the receive queues. We extend this approach to suggest a few bandwidth management configurations on the transmit queue with the same queue-id as used by the receive side hardware filter. It uses the information from the application flow requirements to suggests different bandwidth management rates for each hardware filter configuration.

```
   Flow (sip:sport, dip:dport, UDP, 2Gbps) ->
[
 { 5T(sip:sport, dip:dport, UDP, Q0), TxRL(2Gbps -> Q0)},
 { 5T(sip:sport, dip:dport, UDP, Q0), TxRL(1Gbps -> Q0)},
 { 5T(sip:sport, dip:dport, UDP, Q0), TxRL(None  -> Q0)},
 { 5T(sip:sport, dip:dport, UDP, Q1), TxRL(2Gbps -> Q1)},
 { 5T(sip:sport, dip:dport, UDP, Q1), TxRL(1Gbps -> Q1)},
 { 5T(sip:sport, dip:dport, UDP, Q1), TxRL(None  -> Q1)},
 { 5T(sip:sport, dip:dport, UDP, Q2), TxRL(2Gbps -> Q2)},
 { 5T(sip:sport, dip:dport, UDP, Q2), TxRL(1Gbps -> Q2)},
 { 5T(sip:sport, dip:dport, UDP, Q2), TxRL(None  -> Q2)}
]
```

Figure 6.6: Example of the configuration suggestions generated the oracle

Currently we implement a simple strategy of generating configuration suggestions by mapping the given flow to every send/receive queue-pair. We suggest a configuration by inserting a hardware filter for receive steering to a particular queue-id, and additional bandwidth management configurations with different rates for the same queue-id. The different rates used are in `1Gbps` decrements starting from the flow requested rates.

Figure 6.6 is an example of the configuration suggestions generated by the oracle based on an input of a single UDP flow (e.g. `Flow (sip:sport, dip:dport)`) with a desired bandwidth requirement of `2Gbps`, and three available send/receive queue-pairs.

The oracle accepts the flow and suggests a set of configuration changes which includes the five-tuple receive filters (shown with `5T(...)`) directing the given flow to each of the available queues, and hardware rate-limiters (shown with `TxRL(...)`) specifying bandwidth limits on the queues used by the hardware filters based on the desired rate of the flow. In this example, the oracle has suggested nine relevant configuration changes based on the information in the flow for further analysis.

This example illustrates how the oracle can use the available information to generate the most interesting hardware configuration suggestions, and hence reduce the overall search-space.

Evaluating resource allocation plans: Cost functions

Cost functions provide a way to evaluate and score different resource allocation plans based on high-level system policies. In the context of providing policy-based bandwidth management, cost functions provide us a way to tell which bandwidth allocation plan is more desirable for the current policy.

The cost function for scoring queue allocation works by taking the potential mappings from flows to queues and then scoring these mappings based on the configured policy. We extend the cost function from section 5.4.4 of previous chapter to use additional information about bandwidth rates associated with queues and flows. This additional information allows us to write bandwidth allocation policies in addition to hardware queue allocation policies.

We evaluate our system with two different cost functions. We use a bandwidth balancing cost function which tries to balance the available bandwidth among all the flows, and a priority based cost function which tries to meet the requirements of high priority flows for the bandwidth allocation, and then balances any remaining bandwidth between all other flows.

The implementation of the priority based cost function is extension of Alg. 1, and we present a high-level sketch of the modified cost function in Alg. 3. Our modifications add additional way of measuring the cost based on difference in the rate limit applied to each queue and the aggregate bandwidth required by all the flows mapped to that queue. We calculate separate costs for queues with high priority flows and for queues with best effort flows. We also include additional penalty for not meeting the bandwidth requirements of high priority flows.

Finding the resource allocation plan: Greedy search

The exploration of the NIC configuration space to find a good configuration for the current application requirements is achieved by using greedy search. The search works by finding a best configuration for a single flow at a time. For each flow it will use the hardware oracle to find a set of the most

Algorithm 3: Cost function for static bandwidth management

Input : The available queues Qs and flows F
Input : K queues assigned to HP flows
Input : A function `isHP()` to determine if a flow is HP
Input : The flow mapping $fmap$
Input : Bandwidth requirements of flows $fmapBW$
Input : Rate limits per queue $qmapRL$
Output : A cost value

```
// determine HP and BE flows
```
$(F_{HP}, F_{BE}) \leftarrow$ partition F with `isHP()` function
```
// determine queues for each flow class
```
$Qs_{HP} \leftarrow$ the first K queues from Qs
$Qs_{BE} \leftarrow$ the remaining queues after K are dropped from Qs
```
// are flows assigned to the proper queues?
```
$OK_{HP} \leftarrow \forall f \in F_{HP} : fmap[f] \in Qs_{HP}$
$OK_{BE} \leftarrow \forall f \in F_{BE} : fmap[f] \in Qs_{BE}$
if *(not* OK_{HP}*) or (not* OK_{BE}*)* **then**
  return *CostReject 1*
```
// Find difference in rate limit and BW
   requirements of all flows mapped to queue
```
$Diff(q) \leftarrow qmapRL[q] - sum(fmapBW[f]) \forall f \in fmap :$
$fmap[f] == q$
```
// Calculate cost based on BW difference for all BE
   flows
```
$cost_{BE} \leftarrow abs(sum(Diff(q))) \forall q \in Qs_{BE}$
```
// Calculate cost based on BW difference for all HP
   flows
```
$cost_{HP} \leftarrow abs(sum(Diff(q))) \forall q \in Qs_{HP}$
return *CostAccept* $(HPPenalty * cost_{HP}) + (cost_{BE})$

interesting hardware configurations, and then score them using a cost function to find the best configuration for the current flow. Then the search will incrementally evaluate the next flow. The greedy search approach we use for smart queue management also works for managing the bandwidth controllers. We only had to extend the search to support multiple configuration changes for each step in the greedy flow instead of just one configuration change as assumed in the smart queue management solution. The multiple configuration changes are needed to incorporate the configuration of hard-

ware filters and the bandwidth controller.

Implementing selected resource allocation plan: Embedding

After finding the best resource allocation plan, this step takes the resource allocation plan found by the search step and implements it by configuring the NIC hardware and by emulating any missing functionality in software. In the context of bandwidth management, this step will be responsible for configuring the bandwidth management hardware and providing a software implementation for bandwidth control when no suitable hardware capability is available.

Dragonet can implement this step of configuring the PRG graph using the selected configuration, and then embedding the configured PRG in the LPG graph representing the current network stack state and flows. This embedding can use boolean logic to work out which missing functionalities need to be emulated in software.

Implementing the software bandwidth controller: Dragonet needs a way of emulating bandwidth management capabilities in software when hardware support is not available. For this we implemented the simple token bucket based bandwidth controller in Dragonet and configured it to behave as a simple rate controller. We configured out bandwidth controller to mimic the behavior of the hardware rate controllers of the Intel i82599 NIC by setting the burst size and bucket size to the provided bandwidth rate.

A typical bandwidth controller implementation involves a queue as it may have to hold onto the packet for a while before transmitting it. We implemented the software bandwidth controller by re-using the communication queue between the application and the Dragonet stack endpoint. This simplifies the packet processing and the implementation of bandwidth controllers by removing the need for an additional queue but at the cost of some flexibility. This reduced flexibility shows in the form of software bandwidth control happening at the granularity of an application endpoint, and not at the granularity of a socket endpoint.

The current implementation of our software bandwidth controller and its location in the application-stack communication queue can be easily replaced with other implementations which do not have the limitations outlined above. As the lack of flexibility in our current software bandwidth management implementation does not hinder us from evaluating the automatic configuration of bandwidth controllers, we use it in our initial use cases. We plan to extend it in the future with a more flexible implementation.

Current limitations: Support for the automatic embedding of a bandwidth controller node is currently missing, and we use hard-coded configurations in our benchmarks. Note that the configurations were correctly generated by the search step, and we use these auto-generated configurations to hard-code the parameters for the bandwidth controllers.

6.5.2 Evaluation

This section presents the evaluation of the bandwidth management capabilities of Dragonet. We focus on showing that:

- bandwidth management in Dragonet works.
- Dragonet generates reasonable resource allocation plans based on policies in the cost function.
- Dragonet provides predictable and desired behavior based on the user policies specified in the form of cost functions.

Experiment setup

We use the same hardware setup as described in section 6.4.2. In addition to the Intel i82599 10GbE NIC we also use the Solarflare SFC9020 10GbE NIC on the server to as a representative of a NIC without bandwidth management support in the hardware to run our experiments. This allows us to show how Dragonet behaves on NICs with and without hardware support for traffic bandwidth management.

We are using the same OS and other software as described in section 6.4.2. In addition we use Intel's DPDK [int] and Solarflare's OpenOnload [Sol10a] libraries to enable Dragonet to use advanced features of the i82599 and SFC9020 NICs respectively.

Application details: Based on the `netperf` configurations we use in section 6.4.2 (Quantifying benefits of HW bandwidth controllers), we implement a simple UDP traffic generator which creates a separate connection for each client machine and then continuously sends messages of the specified size, and for the specified duration over each connection. We use a separate thread running on a dedicated core for each connection to avoid conflicts related to CPU sharing. We are currently using `UDP` for this benchmark as `TCP` support in Dragonet is limited.

We monitor traffic received by each client machine to evaluate the bandwidth controlling behavior. As Dragonet uses an additional core and polling for each hardware queue we could not reliably compute the idle CPU time and service demand. Hence we are only reporting the bandwidth observed by each client to show the effectiveness of bandwidth control.

Micro-benchmark details: In order to focus on the impact of bandwidth-managing capabilities, we follow the setup from our previous benchmarks. We use a simple setup with four separate client machines using four connections. We reduce the impact of other parameters by using a separate thread and a separate core for each load generation thread, and we also use a queue allocation plan where each connection will get its own hardware queue from Dragonet. We ensure that all resources and parameters for the workloads are identical, except for the available bandwidth which is controlled based on the configuration generated by Dragonet. This allows us to evaluate the impact of our bandwidth controllers in isolation from other parts of Dragonet.

Dragonet deployment details: We use a Dragonet deployment with five hardware queues and the cost-function of balancing the workload ("balance") and prioritizing the workload ("static"). As this configuration has only four active flows, each flow will get its own set of hardware queues

for sending and receiving. So, the balance and priority cost-functions do provide the same queue allocation plan, but they differ in their bandwidth allocation plans.

Each flow in the micro-benchmark requests `5Gbps` as its desired bandwidth, and one of the flows is marked as a high priority flow. We configure our cost functions to only use a maximum of `8Gbps` total among the flows from the micro-benchmark. This bandwidth restriction ensures that the cost functions will generate bandwidth allocation plans that are similar to the ones when benchmarking the hardware bandwidth controllers in section 6.4.

Configurations used: We use the following configurations in our benchmark:

- **Default:** This configuration does not apply any bandwidth management, and only allocates one hardware queue per flow.
- **Balance (Bal):** This configuration uses a balancing cost function which aims to be fair to all the flows by dividing the provided bandwidth (`8Gbps`) among the flows.
- **Static (Static):** This configuration does static allocation of bandwidth to meet the requirements of high priority flows and then balances the remaining bandwidth among the remaining best effort flows.

For the "Bal" and "Static" configurations, we show results for both using hardware bandwidth controllers ("_HW") and using software bandwidth controllers ("_SW").

For all the configurations we run experiments with each flow active alone, and all flows active together to see the effectiveness of the bandwidth management.

Evaluations and observations

This section presents the results of our benchmarks and provides our observations on them. Figure 6.7 presents the per client bandwidth observed for different bandwidth management policies implemented using cost functions for two NICs.

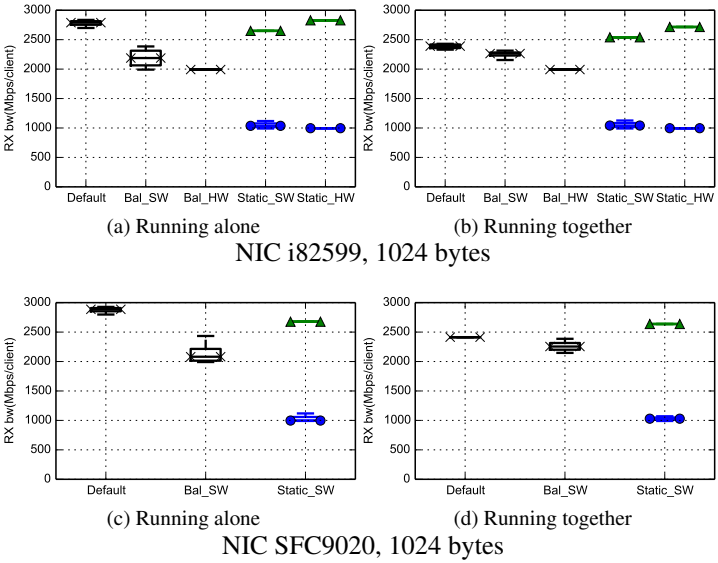


Figure 6.7: Evaluation of UDP bandwidth benchmark on Dragonet

The "Default" configuration represents the results without any bandwidth management. The bandwidth while running each flow alone with this configuration shows that Dragonet can only reach a bandwidth of `2.8Gbps`. This is `42%` less than the `4.8Gbps` bandwidth observed on Linux with a similar setup. We believe that this decrease in bandwidth is due to an implementation issue in the Dragonet data path that we have not resolved yet. In our initial benchmarking of the bandwidth control, we work around this issue by only using bandwidth rates of `2Gbps` and less. As we limit ourselves to data rates reachable by Dragonet, we believe that our observations about the ability to configure and use the hardware bandwidth controllers are valid.

Evaluation of Dragonet resource allocation capabilities: We use two different cost functions implementing different bandwidth allocation policies to evaluate the resource management abilities of Dragonet.

Our balancing cost function aims to be fair to all the flows by dividing the available hardware queues and the bandwidth equally among them, regardless of their desired bandwidth. In our experiment setup of four flows, the balance cost function divides the available `8Gbps` into four flows, giving each flow `2Gbps` even though all of the flows have requested `5Gbps`.

On the other hand, the static cost function aims to prioritize the requirements of the high priority flows. This cost function gives the requested `5Gbps` bandwidth to the high priority flow, and then equally divides the remaining `3Gbps` bandwidth among the best effort flows.

These two example cost functions show that it is possible to write a cost function which will find a desirable resource allocation plan from the available configuration space.

Evaluation of Dragonet bandwidth management implementation: We evaluate Dragonet's capability to use a resource allocation plan to provide bandwidth management in hardware and in software. Once the resource allocation plan is generated, we work around the missing implementation of the embedding step (section 6.5.1) by hard-coding the configuration for the rate controllers in the Intel i82599 NIC, and the software rate controllers when the hardware rate controllers are not available.

The results for the "Bal_HW" and "Static_HW" configurations for Intel i82599 in a figure 6.7 show that Dragonet is able to configure the hardware correctly, and get the expected behavior based on the configurations selected using the respective cost function.

The results for the "Bal_SW" and "Static_SW" configurations for both the Intel i82599 and the Solarflare SFC9020 NICs show that Dragonet can provide bandwidth control in accordance with the given cost function using the software implementation when no hardware capabilities are available.

6.5.3 Observations

We have presented micro-benchmark results on two different NICs with two different cost functions which are used by Dragonet to provide policy-based

resource allocation. Our evaluation is far from complete, and we are still missing the implementation of the embedding step to have a complete solution for bandwidth management in Dragonet. As we have implemented this step for the hardware queue management, we believe that we should be able to provide a complete solution with additional engineering efforts.

With our initial results, we claim that the Dragonet approach of modeling networking hardware to enable reasoning about hardware capabilities, and using cost functions to implement policy-based resource management can be used for bandwidth management.

6.6 Conclusion

In this chapter we argue that bandwidth management is a useful abstraction to provide differentiated performance to applications. Using NIC hardware capabilities for providing bandwidth management also has potential for additional benefits in the form of reduced CPU load. Using the hardware capabilities to provide bandwidth control is difficult due to limited support from the OS, interaction of these features with different part of the network stack and diversity in the level of hardware support for bandwidth management provided by different NIC vendors.

Our initial results show that Dragonet can provide a systematic approach to use these hardware bandwidth controllers while externalizing the resource allocation policies in hardware-agnostic cost functions. Our approach helps the applications using bandwidth control by providing portability across different NIC hardware with different capabilities and makes changing resource allocation policies easier.

Chapter 7

Conclusion

In this thesis, we address the problem of increasing diversity and complexity in NIC hardware. The diversity of the current NIC hardware along with its underlying complexity prevents the applications as well as the host network stack to benefit from the modern NIC hardware capabilities, making this problem increasingly important as the gap between network and CPU speed continues to increase. Part of the difficulty stems from the layered architecture of traditional network stacks which it hides information about the hardware under rigid interfaces to improve the portability. This lack of a systematic way to expose information about evolving hardware capabilities through the network stack makes it difficult for applications to benefit from them without bypassing the OS and using vendor-specific interfaces. Further, the layered architecture and vendor-specific interfaces also makes it difficult to manage limited NIC hardware resources in the presence of competing applications.

The goal of this thesis is to provide a systematic approach to manage and exploit NIC hardware capabilities to the benefit of applications in a portable way. We believe that a hardware-agnostic way to represent packet processing capabilities and flexible interfaces to share the information regarding these capabilities are needed to effectively use the NIC hardware resources.

We approach this problem by making the observation that most packet pro-

cessing capabilities in the modern NICs are packet manipulation functions with configurable behavior. We make the assumption that the dataflow graph based model captures sufficient information about packet processing capabilities of hardware and network stack state to enable automated reasoning about NIC capabilities in a hardware-agnostic way. In this context, we introduce simple abstractions to represent the packet processing performed by the NIC hardware and the packet processing required by the network stack as a dataflow graph. By using the same abstractions to express packet processing in both hardware and the network stack, we are able to automate reasoning about offloading protocol processing. We have implemented the Dragonet host network stack based on this approach.

we introduce new abstractions based on the dataflow models to enable sharing of fine-grained information about packet processing between the network stack layers which can be used for adapting packet processing based on the work done by other layers. We have restructured the network stack to separate resource management from the control path and packet processing. Furthermore, we have implemented the hardware-agnostic and policy-based resource management layer using the information exposed by these interfaces about the NIC capabilities and current packet processing requirements.

Dragonet also uses the dataflow model of packet processing for the runtime assembly of its application-specific library network stack to complement the currently configured NIC hardware capabilities. This adaptability of the software network stack allows dynamic re-allocation of NIC hardware resources at runtime based on high-level policies and changing application requirements.

We demonstrate that using our approach the NIC hardware resources (e.g., NIC hardware queues, packet filters and hardware rate controllers) can be translated into application performance improvement based on the high-level policies.

In the interest of building a working system in the time available, we have not explored the implications of our approach for many interesting aspects of the network stack implementation. For example, our implementation currently does not provide a way to model memory and buffer usage. Modeling memory usage can be instrumental for reasoning about memory pressure,

and managing memory bandwidth. In addition, we have not fully explored how our approach can be adapted to latency sensitive hardware capabilities and workloads, as we have mostly focused on QoS and bandwidth management in our current work. We also currently lack a full implementation of the TCP protocol. We explored the feasibility of modeling TCP, and realized that we need additional support to model timer events as input and output to our models. We also need a safe way to provide read/write access to the protocol-specific state when the receive and send side of the TCP connection are deployed on separate protocol threads. Nevertheless, based on our partial models for the TCP protocol, we believe that our abstractions can be further extended to create a fine-grained model of TCP by capturing timer events as a new type of input for our model.

In addition to implementation limitations, our approach has some other limitations due to the use of the dataflow model. This model does not naturally map to fully programmable hardware (e.g, NIC with onboard programmable core or FPGA) because of our assumption that the NIC hardware has configurable but otherwise fixed functionalities. Hence, our approach may not be able to exploit the full capabilities of such hardware. However, the model we present in this thesis is still useful as it provides a way to use the programmable hardware by modeling a few commonly used functionalities as configurable/loadable functions. Additionally, our model does not capture the exact packet processing (e.g., our model only captures that a checksum is calculated, but does not specify how exactly it is calculated), but assumes certain semantic knowledge about packet processing. This design decision has certain implications, including limitations in pushing custom application logic into the NIC. We discuss a potential extension to support custom application logic in the next section.

7.1 Future work

There are many potential use cases which can benefit from using the dataflow model as a basis for representing packet processing and are worth further exploration.

7.1.1 Pushing application logic into NICs

Our reasoning for offloading packet processing is currently limited to pushing network stack processing into the NIC. However, with the increasing availability of programmable NICs (either with onboard core or FPGA), our model can be extended to support offloading application logic.

Our dataflow model can provide the basis for representation and the interface. The representation can be used to capture application specific packet processing, and the interface can be used to communicate among different layers of the network stack. Supporting such offloading of application logic will require further development of our model to include the detailed description of the packet processing, and we can leverage existing research (e.g. FlexNIC [KPS+16], P4 [BDG+14]) for this purpose. The proposed exploration will also involve extending our current mechanisms of label-based and predicate-based reasoning to use information about processing happening in each node.

The ability to express generic packet processing will allow us to extend Dragonet to manage other accelerators as well. For instance, PacketShader [HJPM10] explored the use of GPUs for performing routing decisions, and we can extend Dragonet to manage GPUs for performing well-formulated network operations (e.g., network routing, checksum calculation).

7.1.2 Integrating with SDN capabilities

Recent progress in *Software Defined Networking* (SDN) [MAB+08] aims to provide software controlled function from the network. Also, there is a push to extend the *OpenFlow* specification [spe12] further to include more packet processing capabilities and protocol oblivious processing [Son13]. This trend towards having software controlled and flexible packet processing capabilities in the network gives an opportunity to use the Dragonet approach and extend it further. For instance, Dragonet can include the packet processing capabilities of the network for optimizing packet processing within a host. Dragonet can also help in end-to-end resource management and packet processing optimizations by providing a handle on host-based NIC capabilities and packet processing, which can be used by the

data-center wide resource management service for end-to-end co-ordination of resources for the network flows.

7.1.3 Integrating with high-level language based approaches

The MirageOS [MS13] explores the use of a high-level, safe language (OCaml) to implement and aggressively optimize the packet processing needed for the application. Currently, this approach focuses on optimizations based on the application requirements but does not take into account the capabilities of the NIC hardware. We believe that there is scope for additional optimizations if this approach has more information about the NIC packet processing capabilities.

7.2 Concluding remarks

As the diversity of the hardware increases, we need to question the current approaches. This thesis has revisited the host network stack architecture and shown the feasibility of using dataflow models to handle the diversity in the NIC hardware. The dataflow model presents an abstraction to specify the protocol processing functionalities, and hence can be used to communicate the packet processing requirements and capabilities in a hardware-agnostic way. The insights and knowledge gained from this exploration provide a foundation for further research in network stack architecture and NIC hardware design.

Bibliography

[Ang01] Boon S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. Technical report, Computer Systems and Technology Laboratory, HP Laboratories, 2001.

[BALL91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, May 1991.

[BBM+12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, 2012.

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[BGK+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[BK03] M. Burnside and A.D. Keromytis. Accelerating application-level security protocols. In *The 11th IEEE International Conference on Networks (ICON)*, pages 313–318. IEEE, 2003.

[BLS12] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: a highly configurable aspect-oriented IP stack. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 435–448. ACM, 2012.

[BPK+14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, October 2014.

[BPR+11] Andrew Baumann, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. Barrelfish specification. Technical report, ETH Zurich, July 2011.

[Bra12] Brandeburg, Jesse and Fastabend, John. Hardware rate limiting control. Presented in Linux Plumbers Conference, August 2012.

[Bro08] Broadcom. *Standard Broadcom NetXtreme II Family Highly Integrated Media Access Controller*, October 2008. Revision PG203-R.

[BSR+06] Willem De Bruijn, Asia Slowinska, Kees Van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos. Safecard: a gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection*, 2006.

[Cav13] Cavium INC. Cavium LiquidIO Server Adapter Family Product Brief. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf, 2013.

[CLW+13a] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An fpga memcached appliance. In *Proceedings*

of the ACM/SIGDA international symposium on Field programmable gate arrays, pages 245–254. ACM, 2013.

[CLW+13b] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA memcached appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 245–254, New York, NY, USA, 2013. ACM.

[Col] Gregory Collins. Hackage: The hashtables package. https://hackage.haskell.org/package/hashtables. Retrieved Nov 2013.

[Dan13] Danga Interactive. Memcached - a distributed memory object caching system. http://memcached.org/, 2013.

[Dat13] Data Differential. memaslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memaslap.html, 2013.

[dcb] Data center bridging task group. http://www.ieee802.org/1/pages/dcbridges.html. Retrieved June 2015.

[dcb09] IEEE Standard for Local and Metropolitan Area Networks–Station and Media Access Control Connectivity Discovery. *IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005)*, pages 1–204, Sept 2009.

[Dcb11a] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)*, pages 1–40, Sept 2011.

[dcb11b] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 18: Enhanced Transmission Selection for Bandwidth Sharing Between Traffic Classes. *IEEE Std 802.1Qaz-2011 (Amendment to IEEE Std*

802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011, IEEE Std 802.1Qbc-2011, and IEEE Std 802.1Qbb-2011), pages 1–110, Sept 2011.

[DEA+09] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, volume 9, Big Sky, Montana, USA, 2009.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[dpd] DPDK development mailing list. Retrieved Feb 2014.

[dst] *dstat: Versatile resource statistics tool*. Retrieved Oct 2015.

[DYL+10] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. In *The 16th International Symposium on High-Performance Computer Architecture*, Bangalore, India, January 2010.

[EBSA+12] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3), May 2012.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[Erw01] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, September 2001.

[eth] ethtool - utility for controlling network drivers and hardware. https://www.kernel.org/pub/software/network/ethtool/. Retrieved Oct 2013.

[eth97] IEEE standards for local and metropolitan area networks: Supplements to carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications - specification for 802.3 full duplex operation and physical layer specification for 100 mb/s operation on two pairs of category 3 or better balanced twisted pair cable (100BASE-T2). pages 0_1–324, 1997.

[FA09] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *29th IEEE International Conference on Distributed Computing Systems*, 2009.

[Fas13a] Fastabend, John. [RFC] net: add a rate_limit attribute to netdev_queue and a rtnetlink. http://patchwork.ozlabs.org/patch/258063/, July 2013.

[Fas13b] Fastabend, John. [RFC PATCH] net: add a tx_queue attribute rate_queue_limits in Mbps. http://lists.openwall.net/netdev/2013/06/24/11, June 2013.

[FBB+05] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. Performance characterization of a 10-Gigabit ethernet TOE. In *Proceedings of 13th IEEE Symposium on High Performance Interconnects*, 2005.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[GAK12] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6), 2012.

[Gil74] Kahn Gilles. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[GMB+15] Jana Giceva, Darko Makreshanski, Claude Barthels, Alessandro Dovis, and Gustavo Alonso. Rack-scale data processing system. In *Second International Workshop on Rack-scale Computing (WRSC 2015)*, April 2015.

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16. Springer, 1998.

[HdB13] Tom Herbert and Willem de Bruijn. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, May 2013.

[HIT05] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32nd Annual IEEE International Symposium on Computer Architecture (ISCA)*, 2005.

[HJP+15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report, UCB Technical Report No. UCB/EECS-2015, 2015.

[HJPM10] S. Han, K. Jang, K.S. Park, and S. Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4), 2010.

[HKM+13] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[HMCR12] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[int] Intel DPDK: Data Plane Development Kit. http://www.dpdk.org/. Retrieved Feb 2014.

[Int03] Intel Corporation. *Intel IXP2400/IXP2800 Network Processor Programmer's Reference Manual*, November 2003.

[Int09] Intel Corporation. *Assigning Interrupts to Processor Cores using an Intel 82575/82576 or 82598/82599 Ethernet Controller*, September 2009. Application Note.

[Int10a] Intel Corporation. *Intel 82576 Gigabit Ethernet Controller Datasheet*, December 2010. Revision 2.61.

[Int10b] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6.

[Int12] Intel. Intel Data Direct I/O Technology (Intel DDIO): A Primer. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, Feb 2012.

[Int13] Intel corporation. Intel 10 Gigabit Linux driver. https://www.kernel.org/doc/Documentation/networking/ixgbe.txt, August 2013.

[ipt] HOWTO maximise download speed via outbound traffic shaping. http://phix.me/dm/. Retrieved October 2015.

[ISAV16] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, Santa Clara, CA, March 2016. USENIX Association.

[JB14] Claudio Föllmi Jeremia Bär. Bulk transfer over shared memory. Master's thesis, ETH Zurich, February 2014.

[JCKL11] Hankook Jang, Sang-Hwa Chung, Dong Kyue Kim, and Yun-Sung Lee. An efficient architecture for a TCP offload engine based on hardware/software co-design. *Journal of Information Science and Engineering*, 509, 2011.

[JJ09] Hye-Churn Jang and Hyun-Wook Jin. Miami: Multi-core aware processor affinity for TCP/IP over multiple network interfaces. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*, HOTI '09, pages 73–82, Washington, DC, USA, 2009. IEEE Computer Society.

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[JWJ+14] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 489–502, Seattle, WA, April 2014.

[Kau14] Antoine Kaufmann. Efficiently executing the Dragonet network stack. Master's thesis, ETH Zurich, October 2014.

[KHM09] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of direct cache access on multi-core systems and 10GbE. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[KPS+16] Antoine Kaufmann, Simon Peter, Navven Kumar Sharma, Thomas Anderson, and Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016.

[KR06] H. Kim and S. Rixner. TCP offload through connection handoff. *ACM SIGOPS Operating Systems Review*, 40(4), 2006.

[KSKR15] Kornilios Kourtis, Pravin Shinde, Antoine Kaufmann, and Timothy Roscoe. Intelligent NIC queue management in the dragonet network stack. In *Proceedings of the 3rd Conference of Timely Results in Operating Systems*, TRIOS'15, 2015.

[LB08] G. Likely and J. Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, 2008.

[LGL+11] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, volume 11, pages 2–2, 2011.

[LGM+12] J.W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *Proceedings of the 20th Annual Symposium on High-Performance Interconnects (HOTI)*, 2012.

[lin12] Linux advanced routing and traffic control. http://lartc.org/lartc.html, 2012.

[LJFK86] Samuel J Leffler, William N Joy, Robert S Fabry, and Michael J Karels. Networking implementation notes, 4.3 BSD edition. Technical report, CSRG/CSD/EECS, University of California, Berkeley, 1986.

[LMW+07] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *IEEE International Conference on Microelectronic Systems Education, 2007. MSE'07*, pages 160–161. IEEE, 2007.

[LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. *SIGPLAN Not.*, 29(6):24–35, June 1994.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[Mai07] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, September 2009. Version 2.2.

[Mic] Microsoft corporation. Scalable networking. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff570736%28v=vs.85%29.aspx`. Retrieved January 2013.

[Mic11] Microsoft Corporation. Information about the TCP chimney offload, receive side scaling, and network direct memory access features in windows server 2008. Microsoft Knowledge Base article 951037, `http://support.microsoft.com/kb/951037`, revision 7.0, February 2011.

[Mim10] Mims, Christopher. Why CPUs Aren't Getting Any Faster. *MIT Technology Review*, Oct 2010.

[Mog03] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th International Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, 2003.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: an IDL for hardware programming. In *4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, 2000.

[MS13] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[neta] Linux kernel documentation: net-prio.txt. Retrieved October 2015.

[Netb] NetFPGA. http://netfpga.org/. Retrieved Aug 2013.

[netc] netperf 2.6.0. http://www.netperf.org/netperf/. Retrieved Aug 2013.

[Netd] Network-based Computing Laboratory, The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/overview/mvapich2/. Retrieved April 2015.

[Net04] Network Working Group. RFC 3720: Internet Small Computer Systems Interface (iSCSI). http://tools.ietf.org/html/rfc3720, 2004. Retrieved October 2015.

[Net08] Netronome Systems, Inc. *Netronome Flow Driver: Programmer's Reference Manual*, 2008. v.1.3.

[OOC07] Pat O'Neil, E O'Neil, and Xuedong Chen. The star schema benchmark. https://www.percona.com/docs/wiki/_media/benchmark%3Assb%3Astarschemab.pdf, 2007.

[Ozd12] Recep Ozdag. Intel ethernet switch FM6000 series-software defined networking. *White Paper*, 2012.

[PCC+14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[PCI10] PCI-SIG. Single Root I/O Virtualization and Sharing Specification. http://www.pcisig.com/specifications/iov/, January 2010.

[PF01] Ian Pratt and Keir Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *IN PROCEEDINGS OF IEEE INFOCOM*, pages 67–76, 2001.

[PLZ+14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, October 2014.

[POS93] *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 1)*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1993.

[PPA+09] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[PSZM12] A. Pesterev, J. Strauss, N. Zeldovich, and R.T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (Eurosys)*, 2012.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[RCK+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite.

In *22nd ACM Symposium on Operating Systems Principles*, pages 73–86, October 2009.

[RDM] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/. Retrieved October 2013.

[Red] Red Hat. Drools - Business Rules Management System. http://drools.org/. Retrieved Aug 2013.

[RGJ+14] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, Seattle, WA, 2014. USENIX.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231, June 1985.

[Ros13] Timothy Roscoe. Inter-dispatcher communication in barrelfish. Technical report, ETH Zurich, March 2013.

[RS07] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th ACM International Symposium on High Performance Distributed Computing (HPDC)*, Monterey, California, USA, 2007.

[SAB+15] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 36–43. IEEE, 2015.

[San14] Sandia National Laboratories. The Portals 4.0.2 Network Programming Interface. http://www.cs.sandia.gov/Portals/portals4.html, Oct 2014.

[SC03] Piyush Shivam and Jeffrey S. Chase. On the elusive benefits of protocol offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, Karlsruhe, Germany, 2003.

[SC08] Inc. Solarflare Communications. OpenOnLoad: high performance network stack. http://www.openonload.org/, 2008.

[SFR04] W Richard Stevens, Bill Fenner, and Andrew M Rudoff. *UNIX network programming*, volume 1. Addison-Wesley Professional, 2004.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[SKKR13] Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, and Timothy Roscoe. Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 3:1–3:6, 2013.

[SKRK13] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *14th Workshop on Hot Topics in Operating Systems*, May 2013.

[soc08] *setsockopt(2): get and set options on sockets - Linux man page*, 2008. Retrieved October 2015.

[soc13] *socket(7) - Linux manual page*, 2013. Retrieved October 2015.

[Sol10a] Solarflare Communications, Inc., 9501 Jeronimo Road, Irvine, California 92618. *Onload User Guide*, 2010. Version 20101221.

[Sol10b] Solarflare Communications, Inc. *Solarflare SFN5122F Dual-Port 10GbE Enterprise Server Adapter*, 2010.

[Sol12] Solarflare Communications, Inc. *SFA6900 ApplicationOnload Engine Overview*, 2012.

[Sol13] Solarflare Communications, Inc. *SFA6902F Dual-Port 10GE SFP+ ApplicationOnload Engine*, 2013. Product brief.

[Son13] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[spe12] Openflow switch specification v1.3.0, 2012.

[ST93] J. M. Smith and C. B. S. Traw. Giving applications access to gb/s networking. *IEEE Network*, 7(4):44–52, July 1993.

[STJP08] J.R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the USENIX'08 Annual Technical Conference*, 2008.

[The09] The Linux Foundation. TCP Offload Engine (toe). `http://www.linuxfoundation.org/collaborate/workgroups/networking/toe`, November 2009. Retrieved January 2013.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[WWL05] W.F. Wang, J.Y. Wang, and J.J. Li. Study on enhanced strategies for TCP/IP offload engines. In *Proceedings of 11th IEEE International Conference on Parallel and Distributed Systems*, volume 1, 2005.

[xAp13] xAppSoftware Blog. How to configure the TCP offload engine(TOE) in Linux. http://www.xappsoftware.com/wordpress/2013/03/25/how-to-configure-the-tcp-offload-engine-toe-in-linux, Mar 2013. Retrieved Feb 2016.

[ZACM14] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, (5):32–41, 2014.

[Zim80] Hubert Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.