Title
System Design for Software Packet Processing

Author
Han, Sangjin

Publication Date
2019

Peer reviewed|Thesis/dissertation

System Design for Software Packet Processing

By

Sangjin Han

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sylvia Ratnasamy, Chair
Professor Scott Shenker
Professor Dorit Hochbaum

Summer 2019

System Design for Software Packet Processing

Abstract

System Design for Software Packet Processing

by

Sangjin Han

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

The role of software in computer networks has never been more crucial than today, with the advent of Internet-scale services and cloud computing. The trend toward software-based network dataplane—as in network function virtualization—requires software packet processing to meet challenging performance requirements, such as supporting exponentially increasing link bandwidth and microsecond-order latency. Many architectural aspects of existing software systems for packet processing, however, are decades old and ill-suited to today's network I/O workloads.

In this dissertation, we explore the design space of high-performance software packet processing systems in the context of two application domains. First, we start by discussing the limitations of BSD Socket, which is a de-facto standard in network I/O for server applications. We quantify its performance limitations and propose a clean-slate API, called MegaPipe, as an alternative to BSD Socket. In the second part of this dissertation, we switch our focus to in-network software systems for network functions, such as network switches and middleboxes. We present Berkeley Extensible Software Switch (BESS), a modular framework for building extensible network functions. BESS introduces various novel techniques to achieve high-performance software packet processing, without compromising on either programmability or flexibility.

To my family, friends, and advisors who made all of this worthwhile.

Contents

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Summary of Contributions . . . 3
- 1.2 Outline and Previously Published Work . . . 4
- 1.3 Research Projects Not Included in This Dissertation . . . 5

2 Background 7
- 2.1 Packet Processing under Multi-Core Environments . . . 7
- 2.2 Virtualization (of Everything) . . . 9
- 2.3 Network Stack Specialization . . . 9

3 MegaPipe: A New Programming Interface for Scalable Network I/O 11
- 3.1 Introduction . . . 11
- 3.2 Motivation . . . 12
  - 3.2.1 Sources of Performance Inefficiency . . . 13
  - 3.2.2 Performance of Message-Oriented Workloads . . . 14
- 3.3 Design . . . 17
  - 3.3.1 Scope and Design Goals . . . 17
  - 3.3.2 Completion Notification Model . . . 18
  - 3.3.3 Architectural Overview . . . 18
  - 3.3.4 Design Components . . . 19
  - 3.3.5 Application Programming Interface . . . 23
  - 3.3.6 Discussion: Thread-Based Servers . . . 24
- 3.4 Implementation . . . 25
  - 3.4.1 Kernel Implementation . . . 25
  - 3.4.2 User-Level Library . . . 26
- 3.5 Evaluation . . . 27
  - 3.5.1 Multi-Core Scalability . . . 27
  - 3.5.2 Breakdown of Performance Improvement . . . 28

3.5.3 Impact of Message Size . . . 28
3.6 Conclusion . . . 29

4 Applications of MegaPipe **30**
4.1 Adopting Existing Server Applications . . . 30
4.1.1 Porting memcached . . . 30
4.1.2 Porting nginx . . . 31
4.2 Macrobenchmark: memcached . . . 31
4.3 Macrobenchmark: nginx . . . 34

5 BESS: A Modular Framework for Extensible Network Dataplane **37**
5.1 Motivation . . . 39
5.1.1 Software-Augmented Network Interface Card . . . 39
5.1.2 Hypervisor Virtual Switch . . . 41
5.1.3 Network Functions . . . 42
5.2 BESS Design Overview . . . 43
5.2.1 Design Goals . . . 43
5.2.2 Overall Architecture . . . 44
5.2.3 Modular Packet Processing Pipeline . . . 45
5.3 Dynamic Packet Metadata . . . 47
5.3.1 Problem: Metadata Bloat . . . 47
5.3.2 Metadata Support in BESS . . . 48
5.3.3 Attribute-Offset Assignment . . . 49
5.4 Resource Scheduling for Performance Guarantees . . . 51
5.5 Implementation Details . . . 53
5.5.1 Overview . . . 54
5.5.2 Core Dedication . . . 54
5.5.3 Pipeline Components . . . 55
5.5.4 BESS Scheduler . . . 56
5.5.5 Packet Buffers and Batched Packet Processing . . . 58
5.5.6 Multi-Core Scaling . . . 59
5.6 Performance Evaluation . . . 61
5.6.1 Experimental Setup . . . 61
5.6.2 End-to-End Latency . . . 61
5.6.3 Throughput . . . 63
5.6.4 Application-Level Performance . . . 63

6 Applications of BESS **65**
6.1 Case Study: Advanced NIC features . . . 65
6.1.1 Segmentation Offloading for Tunneled Packets . . . 65
6.1.2 Scalable Rate Limiter . . . 66
6.1.3 Packet Steering for Flow Affinity . . . 68

6.1.4 Scaling Legacy Applications . . . 69
6.2 Research Projects Based on BESS . . . 70
6.2.1 E2: A Framework for NFV Applications . . . 71
6.2.2 S6: Elastic Scaling of Stateful Network Functions . . . 73

7 Related Work **76**
7.1 MegaPipe . . . 76
7.1.1 Scaling with Concurrency . . . 76
7.1.2 Asynchronous I/O . . . 76
7.1.3 System Call Batching . . . 77
7.1.4 Kernel-Level Network Applications . . . 77
7.1.5 Multi-Core Scalability . . . 77
7.1.6 Similarities in Abstraction . . . 78
7.2 BESS . . . 78
7.2.1 Click . . . 78
7.2.2 High-performance packet I/O . . . 79
7.2.3 Hardware support for virtual network I/O . . . 79
7.2.4 Smart NICs . . . 79
7.2.5 Ideal hardware NIC model . . . 80
7.2.6 Alternative approaches . . . 80

8 Concluding Remarks **81**

List of Figures

2.1 Evolution of AMD and Intel microprocessors in terms of clock frequency and number of cores per processor (log-scaled). The raw data was obtained from Stanford CPUDB [23]. 7
2.2 The parallel speedup of the Linux network stack, measured with a RPC-like workload on a 8-core server. For the experiment we generated TCP client connections, each of which exchanges a pair of 64-byte dummy request and request. 8

3.1 Negative impact of message-oriented workload on network stack performance . . 15
3.2 MegaPipe architecture 19
3.3 Comparison of parallel speedup 27
3.4 Relative performance improvement with varying message size 29

4.1 Throughput of memcached with and without MegaPipe 32
4.2 50th and 99th percentile memcached latency. 33
4.3 Evaluation of nginx throughput for various workloads. 35

5.1 Growing complexity of NIC hardware 40
5.2 BESS architecture 44
5.3 BESS dataflow graph example 45
5.4 Per-packet metadata support in BESS 48
5.5 Example of scope components of metadata attribute 49
5.6 Construction of the scope graph from calculated scope components 50
5.7 Optimal coloring for the scope graph shown in Figure 5.6(b) with three colors. In other words, for the given pipeline, three offsets are enough to accommodate all six attributes with multiplexing. 51
5.8 An example of high-level performance policy and its class tree representation . . 52
5.9 BESS multi-core scaling strategies 59
5.10 Round-trip latency between two application processes. 62
5.11 BESS throughput and multi-core scalability. 63

6.1 System-wide CPU usage for 1M dummy transactions per second 68
6.2 Snort tail latency with different load balancing schemes 70
6.3 Transformations of a high-level policy graph (a) into an instance graph (b, c, d). 72

6.4 Distributed shared object (DSO) space, where object distribution is done in two layers: key and object layers. The key layer partitions the key space; each node keeps track of the current location of every object in the key partition. The object layer stores the actual binary of objects. The key layer provides indirect access to objects, allowing great flexibility in object placement. 74

List of Tables

3.1 List of MegaPipe API functions for userspace applications 22
3.2 Performance improvement from MegaPipe 28

4.1 The amount of code change required for adapting applications to MegaPipe . . 30

5.1 Throughput and CPU usage of memcached in bare-metal and VM environments 64

6.1 TCP throughput and CPU usage breakdown over the VXLAN tunneling protocol 66
6.2 Accuracy comparison of packet scheduling between BESS and SENIC 67

Chapter 1

Introduction

Many design choices of computer networking have been centered around the end-to-end principle [108], which can be roughly summarized as "keep the network simple, and implement application-specific functionality in end hosts." The principle has provided a philosophical basis for the Internet architecture. Intermediary network nodes, such as switches and routers, merely performed simple packet forwarding, *i.e.,* sending datagrams from one end host to another. More sophisticated features, such as reliable file transfer, data encryption, streaming a live video, *etc.*, were supposed to be implemented in end hosts. There is no denying that this separation of roles have greatly contributed to the great success of the Internet. The simple "core" enabled the extraordinary growth of the Internet, as diverse networks were able to join the Internet with ease. Because of its simplicity and statelessness, in-network packet forwarding was easy to be implemented in specialized hardware network processors, enabling the exponential growth of Internet traffic. Beyond providing simple connectivity, the rest—implementation of high-level services—was left to software running on the general purpose host systems at the end. On end nodes, the flexibility and programmability of the software on general-purpose hosts enabled innovative applications, such as World Wide Web.

While the end-to-end principle still stands up as an useful architectural guideline of computer networking, we have seen many trends that do not strictly adhere to the principle. In terms of where functionality should be placed, the boundary between "dumb, fast core in hardware" versus "smart, slow edge in software" is getting blurred. For instance, commercial ISPs are under constant pressure to provide value-added services, such as content distribution and virus scanning, to find a new source of revenue in the highly competitive market [61]. In enterprise networks, the growing need for security and reliability require the network to implement additional network features beyond simple packet forwarding, and hence has led to the proliferation of network middleboxes [113]. These trends have brought sophisticated functionality into the network. On the other hand, the advent of network virtualization and cloud computing has placed much of packet forwarding—VMs or cloud sites—on general-purpose servers in place of dedicated hardware appliances.

As a result, software is playing an increasingly crucial role in network dataplane architecture, with its new domain (network core as well as edge) and new use case (low-level network

functions as well as high-level services). The rise of *software packet processing*[1], which we define as software-based dataplane implementation running on general-purpose processors, can be attributed to two simple factors: necessity and feasibility.

1. **Necessity**: The network dataplane is desired to support new protocols, features, and applications. Software is inherently easier to program and more flexible to adopt new updates. In addition, virtualization decouples of logical services from the underlying physical infrastructure, which need specialized network appliances to be replaced with software running on general-purpose servers.

2. **Feasibility**: The common belief was that software is too slow to be a viable solution for packet processing. The recent progress in software packet processing, however, has made it as a viable approach. Improvements are from both hardware and software. Commodity off-the-shelf servers have greatly boosted I/O capacity with various architectural improvements, such as multi-core processors, integrated memory controllers, and high-speed peripheral bus, to name a few [26]. Combined with efficient implementations of network device drivers for low-overhead packet I/O, software packet processing has shown *potential* for processing 1–100s of Gbps on a single server [24, 34, 41, 103], improving by orders of magnitude over the last decade.

Software packet processing comes with its own challenges to address. First, unlike the common assumptions, being software does not necessarily grant rapid development and deployment with new functionality. In order to introduce a new feature to legacy implementations, such as network stacks in operating system kernel, one has to deal with large and complex existing codebase, convince "upstream" maintainers of its necessity and usefulness, and wait for a new release and its wide-spread deployment. This whole process may easily take several months, if not years. Second, building a high-performance software dataplane is rather like a black art: practiced with experience, rather than principled design; and driven by application-specific, ad-hoc techniques. There do exist specialized and streamlined datapath implementations for various use cases (*e.g.,* [24, 41, 69, 73, 104]), showing that it is possible for software packet processing to achieve high performance. However, a question still remains regarding whether we could achieve the same level of performance with: i) a wider spectrum of applications, beyond specialization; and ii) mature, fully-featured implementations, beyond streamlining.

In this dissertation, we present two systems for high-performance software packet processing, MegaPipe and BESS (Berkeley Extensible Software Switch). While these two systems focus on different layers of the network stack, they both aim to build a general-purpose framework that a variety of applications can be built upon. MegaPipe and BESS also share the same approach to system design; instead of extending the design of an existing system or improve its internal implementation, we took a clean-slate approach to avoid limitations

[1]Here we do not limit the use of the term "packet" for network datagrams. Rather, we use it as an umbrella term for network traffic in general, from the lower, physical/datalink layers all the way up to the application layer in the network stack.

imposed by the legacy abstractions of existing systems. Many design aspects of the existing network stacks were founded decades ago, when the (literally) millionfold increase in network traffic volume and the diverse set of applications of today were beyond imagination. Considering the growing importance of software packet processing, we believe that it is worthwhile to revisit its design space and explore ways to implement a high-performance dataplane, without sacrificing programmability or extensibility.

1.1 Summary of Contributions

We start the first half of this dissertation with MegaPipe, in Chapter 3. The performance of network servers is heavily determined by the efficiency of underlying network I/O API. We examined BSD Socket API, the de-facto standard programming interface for network I/O, and evaluated its impact on network server performance. The API was designed three decades ago, when the processing power of servers were roughly a million times lower and the scale, usage, and application of network servers were far more primitive than today.

We make two research contributions with MegaPipe. Firstly, we identify and quantify the sources of performance inefficiency in BSD Socket API. We show that the API is suboptimal in terms of processor usage, especially when used for the use case that we define "message-oriented workload". In this workload network connections have a short lifespan and carry small messages, as with HTTP, RPC, key-value database servers, etc. Such workloads expose three performance issues of BSD Socket API: 1) system calls are excessively invoked incurring high kernel-user mode switch cost, 2) fie descriptor abstraction imposes high overheads for network connections, and 3) the processor cache usage of the API is not multi-core friendly, causing congestion of cache coherent traffic on modern processors. We conclude that the root cause of these problems is that BSD Socket API was not designed with concurrency in mind, in terms of both workloads (concurrent client connections) and platforms (multi-core processors).

Secondly, we present a new network programming API for high-performance network I/O. In contrast to many existing work by others, MegaPipe takes a clean-slate approach; it is designed from scratch rather than extending the existing BSD Socket API, in order to eliminate fundamental performance limiting factors imposed by its legacy abstractions. Such limiting factors include use of file descriptors for network connections, polling-based event handling, and excessive system call invocation for I/O operations.

We propose "channel" as the key design concept of MegaPipe, which is a per-core, bidirectional pipe between the kernel space and user applications. The channel multiplexes asynchronous I/O requests and event notifications from concurrent network connections. As compared to the BSD Socket API, our channel-based design enables three key optimizations: 1) the channel abstracts network connections as handles, which are much lighter than regular file descriptors; 2) requests and event notifications in the channel can be batched transparently (thus amortizing the cost of system calls), exploiting data parallelism from independent network connections; and 3) since the channel is per-core, the system capacity can

scale linearly without cache contention in multi-core environments. We show that MegaPipe can boost the performance of network servers significantly, in terms of both per-core and aggregate throughput.

In the latter half of this dissertation we present BESS. BESS works at lower layers of the network stack than MegaPipe, mostly on Data Link (L2) and Network (L3) layers in the traditional OSI model. BESS is a modular framework to build dataplane for various types of network functions, such as software-augmented network interface card (NIC), virtual switches, and network middleboxes. BESS is designed to be not only highly programmable and extensible, but also readily deployable as it is backward compatible with existing commodity off-the-shelf hardware. Its low overhead makes BESS suitable for high-performance network function dataplane.

Heavily inspired by Click [65], BESS adopts a modular pipeline architecture based on dataflow graph. BESS can be dynamically configured as a custom datapath at runtime, by composing small components called "modules". Individual modules performs some module-specific packet processing, such as classification, forwarding, en/decapsulation, transformation, monitoring, etc. Each module can be independently extensible and configurable.

BESS itself is neither pre-configured nor hard-coded to provide particular functionality, such as Ethernet bridging or traffic encryption. Instead, the modularity of BESS allows external applications, or controllers, to program their own packet processing datapath by combining modules, built-in or newly implemented. The controller can translate some high-level, application-specific network policy into a BESS dataflow graph in a declarative manner. For example, one can configure BESS to function as an IP router with built-in firewall functionality. Since the datapath can be highly streamlined for specific use cases and modules in the dataflow graph can be individually optimized, BESS is suitable for building a high-performance network dataplane.

BESS incorporates various novel design concepts to enable modular and flexible network functions without compromising on performance. Pervasive batch processing of packets boosts packet processing performance by amortizing per-packet overhead. Dynamic metadata allows modules to exchange per-packet metadata in a safe, efficient, and modular manner. With dynamic metadata, modules can be added and evolve individually, without introducing code bloat or complexity due to functionality coupling. BESS adopts "traffic class" as a first-class abstraction, allowing external controllers to control how processor and network resources are shared among traffic classes. This mechanism enables various service-level objectives to be enforced, such as performance isolation, guarantee, and restriction, all of which are crucial for supporting diverse use cases. We illustrate the details of BESS in Chapter 5.

1.2 Outline and Previously Published Work

The remainder of this dissertation is organized as follows. We introduce two systems that incorporate various novel techniques to support high-performance packet processing at two different layers: application-level upper layer and packet-level lower layer. Chapter 3

demonstrates MegaPipe, a new programming API for higher-layer network I/O operations at network endpoints, and Chapter 4 describes its applications. In Chapter 5 we present BESS, a modular framework to build lower-layer network functions running in-network. Chapter 6 introduces BESS-based research projects in which the author participated. Finally Chapter 8 concludes this dissertation and discusses suggestions for future work.

This dissertation includes previously published, co-authored material with their written permission. The material in Chapter 3 is adopted from [42], and the material in Chapter 5 is based on [40]. Chapter 5 briefly covers work published in [88, 135].

1.3 Research Projects Not Included in This Dissertation

As part of the graduate course, the author also worked on other topics that are not covered here, although they are broadly related to this dissertation under the common theme of high-performance networked systems. These topics and their resulting publications are listed as follows:

- **Expressive programming abstraction for next-generation datacenters** [43]: The traditional servercentric architecture has determined the way in which the existing dataintensive computation frameworks work. Popular largescale computation frameworks, such as MapReduce and its derivatives, rely on sequential access over large blocks to mitigate high interserver latency. The expressiveness of this batch-oriented programming model is limited by embarrassingly parallel operations, which are not applicable to interactive tasks based on random accesses to dataset.

  The low-latency communication in the disaggregated datacenter architecture will enable new kinds of applications that have never been possible in traditional datacenters. In this work we develop and present Celias, a concurrent programming model to fully leverage the "pool of resources" semantics and low endtoend latency of the new datacenter architecture. It provides intuitive yet powerful programming abstractions for large and complex tasks, while still providing elastic scaling and automatic fault tolerance.

- **Network support for resource disaggregation** [35, 39]: Datacenters have traditionally been a collection of individual servers, each of which aggregates a fixed amount of computing, memory, storage, and network resources as an independent physical entity. Extrapolating from recent trends, this research work envisages that future datacenters will be architected in a drastically different manner: all computational resources within a server will be disaggregated into standalone blades, and the datacenter network will directly interconnect them. This new architecture will enable sustainable evolution of hardware components, high utilization of datacenter resources, and high cost effectiveness.

  The unified interconnect is the most critical piece to realize resource disaggregation. As communication that was previously contained within a server now hits the datacenter-wide fabric, resource disaggregation increases load on the network and makes the need

for strict performance guarantees. Our preliminary experiments with latency/bandwidth requirements for popular datacenter workloads indicate that the unified fabric might be within reach, even with current 10G/40G interconnect technologies.

- **Safe and modular programming model for network functions** [89]: The main premise of NFV, moving hardware network functions to virtualized software environment, has proven more challenging than expected. We posit that the root cause of this delay in transition is twofold. First, developing new NFs is a time-consuming process that requires significant expertise. Developers have to rediscover and reapply the same set of optimization techniques, to build a high-performance network function. Today, building NFs lacks what modern data analytics frameworks (e.g., Spark [137] and Dryad [52]) provide: high-level, customizable network processing elements that can be used as building blocks for NFs. Second, the high overheads associated with hardware-provided isolations, VMs and containers, hinders meeting the performance requirements of network service providers and enterprises.

  In this research project, we design and implement Netbricks, a safe and fast framework for network functions. NetBricks is a development framework that enables rapid implementation of new network functions, achieved with high-level abstractions tailored for network function dataplane. At the same time, Netbricks is also a runtime framework that allows various network functions to be safely chained, even if they are from multiple untrusted vendors and tenants, with theoretically minimal overheads. The key contribution of this work is the technique that we named "Zero-Copy Software Isolation" (ZCSI). ZCSI provides packet and state isolation for network functions with no performance penalty, not requiring any special isolation mechanisms from hardware.

Chapter 2

Background

In this chapter, we explore a few notable ongoing trends that are relevant to software packet processing, along with their motivations, implications, and challenges.

2.1 Packet Processing under Multi-Core Environments

Microprocessor design paradigms have changed greatly over the last decades. In the past, processor performance improvement used to be in the form of an exponential increase in clock frequency (Figure 2.1). Software performance improved with increase in clock speed without any extra effort on the part of the developer. The exponential increase in clock frequency stopped in the early 2000s due to physical limits such as power consumption (and therefore

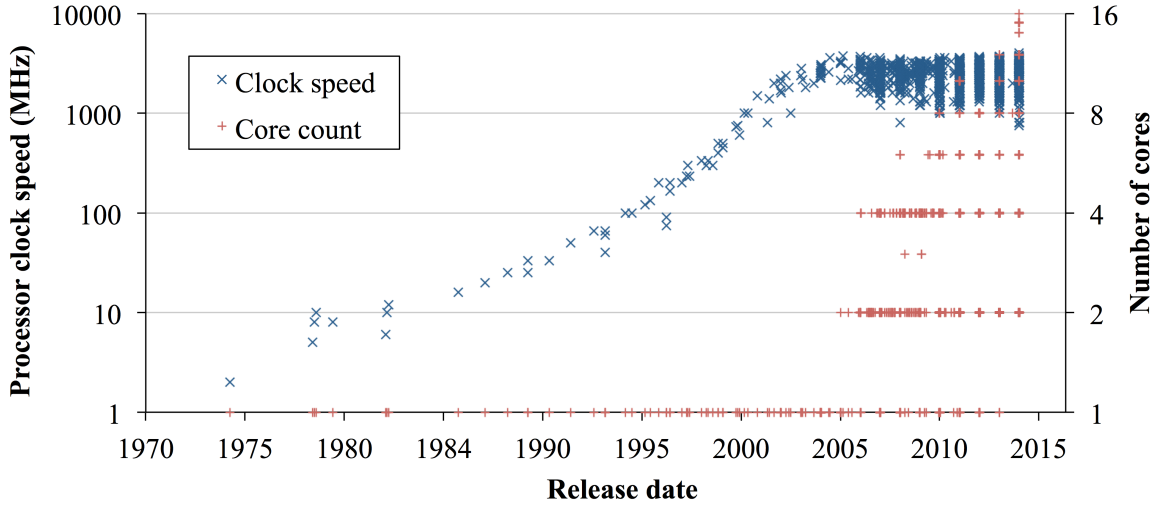


Figure 2.1: Evolution of AMD and Intel microprocessors in terms of clock frequency and number of cores per processor (log-scaled). The raw data was obtained from Stanford CPUDB [23].

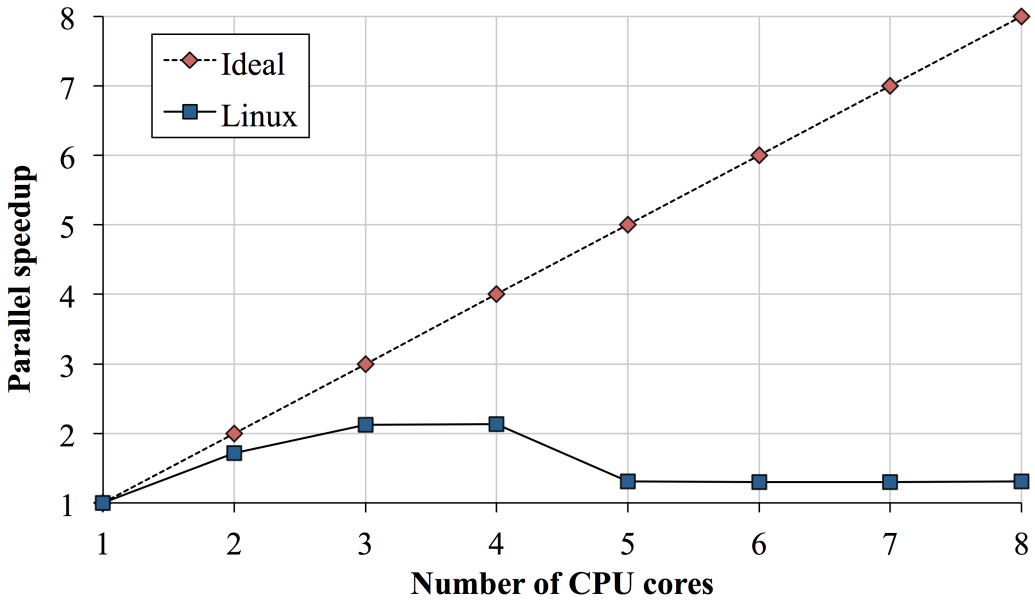


Figure 2.2: The parallel speedup of the Linux network stack, measured with a RPC-like workload on a 8-core server. For the experiment we generated TCP client connections, each of which exchanges a pair of 64-byte dummy request and request.

heat dissipation) and the current leakage problems. Instead, microprocessors started adopting the multi-core architecture, i.e., incorporating multiple independent processing units (cores) in a single processor package, to provide more processing power. Now multi-core processors have become the norm, and today typical high-end servers are equipped with two or four processors, each of which has 4-32 cores as of today.

Any software needs to be fundamentally redesigned in order to exploit the multi-core architecture, and software packet processing is no exception. The network stack must be parallelized to keep pace with increase in network bandwidth, as individual cores are unlikely to get significantly faster in the foreseeable future. The current best practice of network stack parallelization is to duplicate the processing pipeline across cores, ensuring consistent access to shared data structures (*e.g.,* TCP connection states or IP forwarding table) with explicit locks and other synchronization mechanisms [24, 133]. While implementing this scheme is straightforward, its performance is known to be suboptimal due to inter-core synchronization costs, such as serialization and cache coherence traffic [12].

Figure 2.2 shows the non-ideal scaling behavior of the Linux TCP/IP stack. The "Ideal" line represents desired linear scalability, i.e., speedup by a factor of N with N cores. The actual scaling behavior of Linux is far worse than the ideal case, showing only 2.13x speedup with 3 cores at peak. The aggregate throughput collapses as we add more cores. This motivating example clearly indicates that the current architecture cannot effectively exploit multi-core processors. For optimal performance with near-linear scalability, we need to use design principles which minimize inter-core communication in our network stack.

2.2 Virtualization (of Everything)

The basic processing unit of the current datacenter is a virtual machine (or containers, as a lightweight form of server virtualization). With server virtualization, one physical server can be divided into multiple, isolated virtual machines. Each virtual server acts like an independent physical device, being capable of running its own operating system. Server virtualization brings a few obvious advantages. First, the resource utilization of physical resources is inherently higher than that of bare-metal servers, since multiple virtual machines can be consolidated into one server. Second, the physical infrastructure of datacenters can be highly homogeneous; a large array of inexpensive, commodity servers can be utilized to run a wide range of applications under a virtualized environment. Third, in the presence of workload fluctuation, resources can be assigned elastically by adjusting the number of virtual servers, as physical servers and applications running on them are decoupled.

Server virtualization brings a few challenges to the network fabric of datacenters, in order to achieve its full potential. Virtual machine instances basic network connectivity network connectivity, which used to be solely delivered via dedicated physical network appliances. Network virtualization achieves them by creating an abstracted view of virtual network resources from the underlying network infrastructure, consisting of both hardware and software components. This virtualized network must meet not only the requirements of traditional networks, but also ones that are specific to virtualized environments, such as VM mobility, multi-tenancy, and performance isolation.

Network function virtualization (NFV) takes network virtualization one step further, both in its application domain (not only datacenters but also wide-area networks) and its scope (advanced network functions beyond basic network connectivity). NFV aims to virtualize a wide variety of network functions including switches and routers, as well as layer 4-7 middleboxes such as load balancers, firewalls, WAN accelerators, and VPNs, and interconnect them to create a service chain. The transition from proprietary, purpose-built network gears to network functions running on general-purpose computing platforms aims to enable more open, flexible, and economical networking. From the dataplane perspective, NFV adds significant burden to software packet processing. Server virtualization already introduces inherent overheads over bare-metal computing (*e.g.,* IO emulation and VM exit). In addition to that, NFV requires not only fast packet processing in each NF, but also an efficient way to relay packets across NFs.

2.3 Network Stack Specialization

Many recent research projects report that *specialized* network stacks can achieve very high performance: an order of magnitude or two higher than the performance we can expect from general-purpose network stacks, in terms of throughput and/or latency. Examples include software routers [24, 41], web servers [54], DNS servers [74], and key-value stores [60, 69]. We note two common themes for achieving high performance in the design of these

specialized systems. First, they tightly couple every layer of the network stack in a monolithic architecture: from the NIC device driver, through the (streamlined) protocol suite, up to the application logic – which used to be a separate entity in general-purpose systems – itself. This approach is amenable to aggressive cross-layer optimizations. Second, their dataplane implementations bypass the operating system (and its general-purpose network stack) and make exclusive use of hardware resources; i.e., NICs and processor cores.

The performance improvement from these specialized network stacks comes at the cost of generality. One obvious drawback of application-specific optimizations is that they do not readily benefit other existing (thousands of) network applications, unless we heavily modify each application. Also, the holistic approach adopted by the specialized systems rules out the traditional roles of operating systems in networking: regulating use of shared hardware resources, maintaining the global view of the system, (de-)multiplexing data stream across protocols and applications, providing stable APIs for portability, and so on. Many research papers claim that this issue can be avoided by offloading such OS features onto NIC hardware. However, as we will discuss in Chapter 5, it is unrealistic to assume that NIC hardware would provide all functionality required for a variety of applications in a timely manner.

Although the reported performance of specialized network stacks is impressive, we argue that the following question has been overlooked in the systems research community: how much of the speedup of those systems is fundamental to the design (*e.g.,* monolithic architecture with cross-layer optimizations) and how much is from artifacts (*e.g.,* suboptimal implementation of existing general-purpose network stacks)? In other words, is it possible to have a general-purpose network stack with the performance of specialized systems? To answer this question, this dissertation examines 1) how and when are the individual techniques developed for specialized systems are particularly effective? 2) how many of them can be back-ported to general-purpose network stacks? 3) what would be the fundamental trade-off between generality and performance?

Chapter 3

MegaPipe: A New Programming Interface for Scalable Network I/O

3.1 Introduction

Existing network APIs on multi-core systems have difficulties scaling to high connection rates and are inefficient for "message-oriented" workloads, by which we mean workloads with short connections[1] *and/or* small messages. Such message-oriented workloads include HTTP, RPC, key-value stores with small objects (e.g., RAMCloud [86]), etc. Several research efforts have addressed aspects of these performance problems, proposing new techniques that offer valuable performance improvements.

However, they all innovate within the confines of the traditional socket-based networking APIs, by either *i)* modifying the internal implementation but leaving the APIs untouched [20, 91, 119], or *ii)* adding new APIs to complement the existing APIs [10, 30, 31, 68, 134]. While these approaches have the benefit of maintaining backward compatibility for existing applications, the need to maintain the *generality* of the existing API – e.g., its reliance on file descriptors, support for blocking and nonblocking communication, asynchronous I/O, event polling, and so forth – limits the extent to which it can be optimized for performance. In contrast, a clean-slate redesign offers the opportunity to present an API that is specialized for high performance network I/O.

An ideal network API must offer not only high performance but also a simple and intuitive programming abstraction. In modern network servers, achieving high performance requires efficient support for *concurrent I/O* so as to enable scaling to large numbers of connections per thread, multiple cores, etc. The original socket API was not designed to support such concurrency. Consequently, a number of new programming abstractions (e.g., `epoll`, `kqueue`, etc.) have been introduced to support concurrent operation without overhauling the socket API. Thus, even though the basic socket API is simple and easy to use, programmers face the

[1] We use "short connection" to refer to a connection with a small number of messages exchanged; this is not a reference to the absolute time duration of the connection.

unavoidable and tedious burden of layering several abstractions for the sake of concurrency. Once again, a clean-slate design of network APIs offers the opportunity to design a network API from the ground up with support for concurrent I/O.

Given the central role of networking in modern applications, we posit that it is worthwhile to explore the benefits of a clean-slate design of network APIs aimed at achieving both high performance and ease of programming. In this chapter we present MegaPipe, a new API for efficient, scalable network I/O. The core abstraction MegaPipe introduces is that of a *channel* – a per-core, bi-directional pipe between the kernel and user space that is used to exchange both asynchronous I/O requests *and* completion notifications. Using channels, MegaPipe achieves high performance through three design contributions under the roof of a single unified abstraction:

- **Partitioned listening sockets**: Instead of a single listening socket shared across cores, MegaPipe allows applications to clone a listening socket and partition its associated queue across cores. Such partitioning improves performance with multiple cores while giving applications control over their use of parallelism.
- **Lightweight sockets**: Sockets are represented by file descriptors and hence inherit some unnecessary file-related overheads. MegaPipe instead introduces lwsocket, a lightweight socket abstraction that is not wrapped in file-related data structures and thus is free from system-wide synchronization.
- **System Call Batching**: MegaPipe amortizes system call overheads by batching asynchronous I/O requests and completion notifications within a channel.

We implemented MegaPipe in Linux and adapted two popular applications – memcached [**Memcached**] and the nginx [123] – to use MegaPipe. In our microbenchmark tests on an 8-core server with 64 B messages, we show that MegaPipe outperforms the baseline Linux networking stack between 29% (for long connections) and 582% (for short connections). MegaPipe improves the performance of a modified version of memcached between 15% and 320%. For a workload based on real-world HTTP traffic traces, MegaPipe improves the performance of nginx by 75%.

The rest of the chapter is organized as follows. We expand on the limitations of existing network stacks in §3.2, then present the design and implementation of MegaPipe in §5.2 and §3.4, respectively. We evaluate MegaPipe with microbenchmarks and macrobenchmarks in §3.5, and review related work in §7.1.

3.2 Motivation

Bulk transfer network I/O workloads are known to be inexpensive on modern commodity servers; one can easily saturate a 10 Gigabit (10G) link utilizing only a single CPU core. In contrast, we show that message-oriented network I/O workloads are very CPU-intensive

and may significantly degrade throughput. In this section, we discuss limitations of the current BSD socket API (§3.2.1) and then quantify the performance with message-oriented workloads with a series of RPC-like microbenchmark experiments (§3.2.2).

3.2.1 Sources of Performance Inefficiency

In what follows, we discuss known sources of inefficiency in the BSD socket API. Some of these inefficiencies are general, in that they occur even in the case of a single core, while others manifest only when scaling to multiple cores – we highlight this distinction in our discussion.

Issues Common to Both Single-Core and Multi-Core Envronments

- **File Descriptors**: The POSIX standard requires that a newly allocated file descriptor be the lowest integer not currently used by the process [124]. Finding 'the first hole' in a file table is an expensive operation, particularly when the application maintains many connections. Even worse, the search process uses an explicit per-process lock (as files are shared within the process), limiting the scalability of multi-threaded applications. In our `socket()` microbenchmark on an 8-core server, the cost of allocating a single FD is roughly 16% greater when there are 1,000 existing sockets as compared to when there are no existing sockets.

- **System Calls**: Previous work has shown that system calls are expensive and negatively impact performance, both directly (mode switching) and indirectly (cache pollution) [119]. This performance overhead is exacerbated for message-oriented workloads with small messages that result in a large number of I/O operations.

Issues Specific to Multi-Core Scaling

- **VFS**: In UNIX-like operating systems, network sockets are abstracted in the same way as other file types in the kernel; the Virtual File System (VFS) [64] associates each socket with corresponding file instance, inode, and dentry data structures. For message-oriented workloads with short connections, where sockets are frequently opened as new connections arrive, servers quickly become overloaded since those globally visible objects cause system-wide synchronization cost [20]. In our microbenchmark, the VFS overhead for socket allocation on eight cores was 4.2 times higher than the single-core case.

- **Contention on Accept Queue**: As explained in previous work [20, 91], a single listening socket (with its `accept()` backlog queue and exclusive lock) forces CPU cores to serialize queue access requests; this hotspot negatively impacts the performance of both producers (kernel threads) enqueueing new connections and consumers (application threads) accepting new connections. It also causes CPU cache contention on the shared listening socket.

- **Lack of Connection Affinity**: In Linux, incoming packets are distributed across CPU cores on a flow basis (hash over the 5-tuple), either by hardware (RSS [102]) or software (RPS [45]); all receive-side processing for the flow is done on a core. On the other hand, the transmit-side processing happens on the core at which the application thread for the flow resides. Because of the serialization in the listening socket, an application thread calling `accept()` may accept a new connection that came through a remote core; RX/TX processing for the flow occurs on two different cores, causing expensive cache bouncing on the TCP control block (TCB) between those cores [91]. While the per-flow redirection mechanism [50] in NICs eventually resolves this core disparity, short connections cannot benefit since the mechanism is based on packet sampling.

In parallel with our work, the Affinity-Accept project [91] has recently identified and solved the first two issues, both of which are caused by the shared listening socket. We discuss our approach (partitioning) and its differences in §3.3.4. To address other issues, we introduce the concept of lwsocket (§3.3.4, for FD and VFS overhead) and batching (§3.3.4, for system call overhead).

3.2.2 Performance of Message-Oriented Workloads

While it would be ideal to separate the aforementioned inefficiencies and quantify the cost of each, tight coupling in semantics between those issues and complex dynamics of synchronization/cache make it challenging to isolate individual costs.

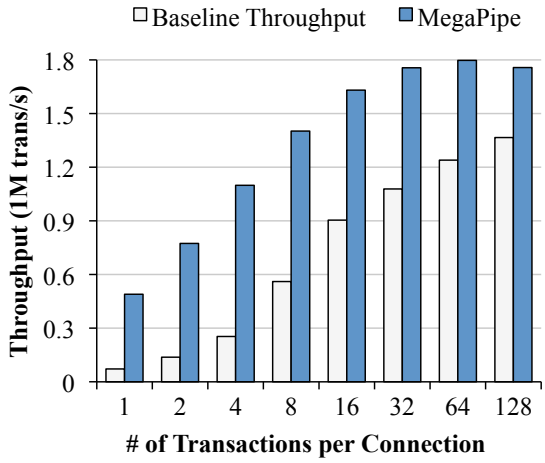
Rather, we quantify their *compound* performance impact with a series of microbenchmarks in this work. As we noted, the inefficiencies manifest themselves primarily in workloads that involve *short connections* or *small-sized messages*, particularly with increasing numbers of CPU cores. Our microbenchmark tests thus focus on these problematic scenarios.

Experimental Setup

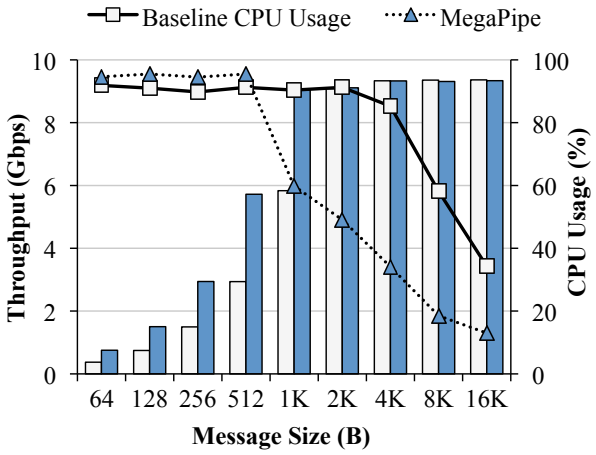
For our tests, we wrote a pair of client and server microbenchmark tools that emulate RPC-like workloads. The client initiates a TCP connection, exchanges multiple request and response messages with the server and then closes the connection.[2] We refer to a single request-response exchange as a *transaction*. Default parameters are 64 B per message and 10 transactions per connection, unless otherwise stated. Each client maintains 256 concurrent connections, and we confirmed that the client is never the bottleneck. The server creates a single listening socket shared by eight threads, with each thread pinned to one CPU core. Each event-driven thread is implemented with `epoll` [30] and the non-blocking socket API.

Although synthetic, this workload lets us focus on the low-level details of network I/O overhead without interference from application-specific logic. We use a single server and

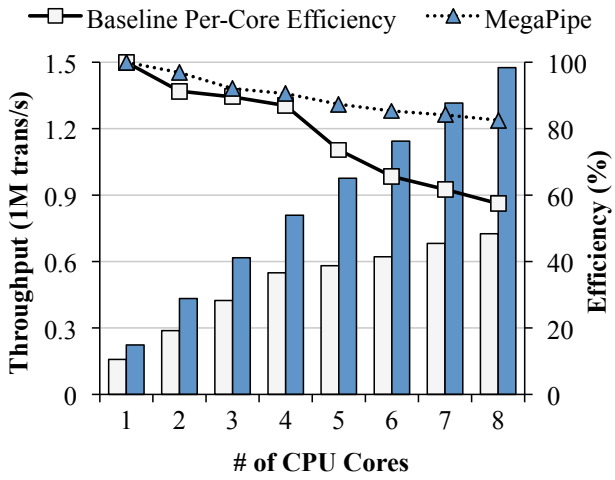
[2] In this experiment, we closed connections with RST, to avoid exhaustion of client ports caused by lingering TIME_WAIT connections.



(a) Connection lifespan



(b) Message size



(c) Number of cores

Figure 3.1: The negative impact in performance from each of varying factor. The default parameters are 64 B messages, 10 transactions per connection, and 8 cores.

three client machines, connected through a dedicated 10G Ethernet switch. All test systems use the Linux 3.1.3 kernel and ixgbe 3.8.21 10G Ethernet device driver [49] (with interrupt coalescing turned on). Each machine has a dual-port Intel 82599 10G NIC, 12 GB of DRAM, and two Intel Xeon X5560 processors, each of which has four 2.80 GHz cores. We enabled the multi-queue feature of the NICs with RSS [102] and FlowDirector [50], and assigned each RX/TX queue to one CPU core.

In this section, we discuss the result of the experiments Figure 3.1 labeled as "Baseline." For comparison, we also include the results with our new API, labeled as "MegaPipe," from the same experiments.

Performance with Short Connections

TCP connection establishment involves a series of time-consuming steps: the 3-way handshake, socket allocation, and interaction with the user-space application. For workloads with short connections, the costs of connection establishment are not amortized by sufficient data transfer and hence this workload serves to highlight the overhead due to costly connection establishment.

We show how connection lifespan affects the throughput by varying the number of transactions per connection in Figure 3.1(a), measured with eight CPU cores. Total throughput is significantly lower with relatively few (1–8) transactions per connection. The cost of connection establishment eventually becomes insignificant for 128+ transactions per connection, and we observe that throughput in single-transaction connections is roughly 19 times lower than that of long connections!

Performance with Small Messages

Small messages result in greater relative network I/O overhead in comparison to larger messages. In fact, the per-message overhead remains roughly constant and thus, independent of message size; in comparison with a 64 B message, a 1 KiB message adds only about 2% overhead due to the copying between user and kernel on our system, despite the large size difference.

To measure this effect, we perform a second microbenchmark with response sizes varying from 64 B to 64 KiB (varying the request size in lieu of or in addition to the response size had almost the same effects). Figure 3.1(b) shows the measured throughput (in Gbps) and CPU usage for various message sizes. It is clear that connections with small-sized messages adversely affect the throughput. For small messages ($\leq$ 1 KiB) the server does not even saturate the 10G link. For medium-sized messages (2–4 KiB), the CPU utilization is extremely high, leaving few CPU cycles for further application processing.

Performance Scaling with Multiple Cores

Ideally, throughput for a CPU-intensive system should scale linearly with CPU cores. In reality, throughput is limited by shared hardware (e.g., cache, memory buses) and/or

software implementation (e.g., cache locality, serialization). In Figure 3.1(c), we plot the throughput for increasing numbers of CPU cores. To constrain the number of cores, we adjust the number of server threads and RX/TX queues of the NIC. The lines labeled "Efficiency" represent the measured per-core throughput, normalized to the case of perfect scaling, where N cores yield a speedup of N.

We see that throughput scales relatively well for up to four cores – the likely reason being that, since each processor has four cores, expensive off-chip communication does not take place up to this point. Beyond four cores, the marginal performance gain with each additional core quickly diminishes, and with eight cores, speedup is only 4.6. Furthermore, it is clear from the growth trend that speedup would not increase much in the presence of additional cores. Finally, it is worth noting that the observed scaling behavior of Linux highly depends on connection duration, further confirming the results in Figure 3.1(a). With only one transaction per connection (instead of the default 10 used in this experiment), the speedup with eight cores was only 1.3, while longer connections of 128 transactions yielded a speedup of 6.7.

3.3 Design

MegaPipe is a new programming interface for high-performance network I/O that addresses the inefficiencies highlighted in the previous section and provides an easy and intuitive approach to programming high concurrency network servers. In this section, we present the design goals, approach, and contributions of MegaPipe.

3.3.1 Scope and Design Goals

MegaPipe aims to accelerate the performance of message-oriented workloads, where connections are short and/or message sizes are small. Some possible approaches to this problem would be to extend the BSD Socket API or to improve its internal implementation. It is hard to achieve optimal performance with these approaches, as many optimization opportunities can be limited by the legacy abstractions. For instance: *i)* sockets represented as files inherit the overheads of files in the kernel; *ii)* it is difficult to aggregate BSD socket operations from concurrent connections to amortize system call overheads. We leave optimizing the message-oriented workloads with those dirty-slate (minimally disruptive to existing API semantics and legacy applications) alternatives as an open problem. Instead, we take a clean-slate approach in this work by designing a new API from scratch.

We design MegaPipe to be conceptually simple, self-contained, and applicable to existing event-driven server applications with moderate efforts. The MegaPipe API provides a unified interface for various I/O types, such as TCP connections, UNIX domain sockets, pipes, and disk files, based on the completion notification model (§3.3.2) We particularly focus on the performance of network I/O in this chapter. We introduce three key design concepts of MegaPipe for high-performance network I/O: partitioning (§3.3.4), lwsocket (§3.3.4), and

batching (§3.3.4), for reduced per-message overheads and near-linear multi-core scalability.

In this work, we mostly focus on an efficient interface between kernel and network applications; we do not address device driver overheads, the TCP/IP stack, or application-specific logic.

3.3.2 Completion Notification Model

The current best practice for event-driven server programming is based on the readiness model. Applications poll the readiness of interested sockets with `select`/`poll`/`epoll` and issue non-blocking I/O commands on the those sockets. The alternative is the completion notification model. In this model, applications issue asynchronous I/O commands, and the kernel notifies the applications when the commands are complete. This model has rarely been used for network servers in practice, though, mainly because of the lack of socket-specific operations such as `accept`/`connect`/`shutdown` (e.g., POSIX AIO [124]) or poor mechanisms for notification delivery (e.g., SIGIO signals).

MegaPipe adopts the completion notification model over the readiness model for three reasons. First, it allows transparent batching of I/O commands and their notifications. Batching of non-blocking I/O commands in the readiness model is very difficult without the explicit assistance from applications. Second, it is compatible with not only sockets but also disk files, allowing a unified interface for any type of I/O. Lastly, it greatly simplifies the complexity of I/O multiplexing. Since the kernel controls the rate of I/O with completion events, applications can blindly issue I/O operations without tracking the readiness of sockets.

3.3.3 Architectural Overview

MegaPipe involves both a user-space library and Linux kernel modifications. Figure 3.2 illustrates the architecture and highlights key abstractions of the MegaPipe design. The left side of the figure shows how a multi-threaded application interacts with the kernel via MegaPipe *channels*. With MegaPipe, an application thread running on each core opens a separate channel for communication between the kernel and user-space. The application thread registers a *handle* (socket or other file type) to the channel, and each channel multiplexes its own set of handles for their asynchronous I/O requests and completion notification events.

When a listening socket is registered, MegaPipe internally spawns an independent accept queue for the channel, which is responsible for incoming connections to the core. In this way, the listening socket is not shared by all threads, but *partitioned* (§3.3.4) to avoid serialization and remote cache access.

A handle can be either a regular file descriptor or a lightweight socket, *lwsocket* (§3.3.4). lwsocket provides a direct shortcut to the TCB in the kernel, to avoid the VFS overhead of traditional sockets; thus lwsockets are only visible within the associated channel.

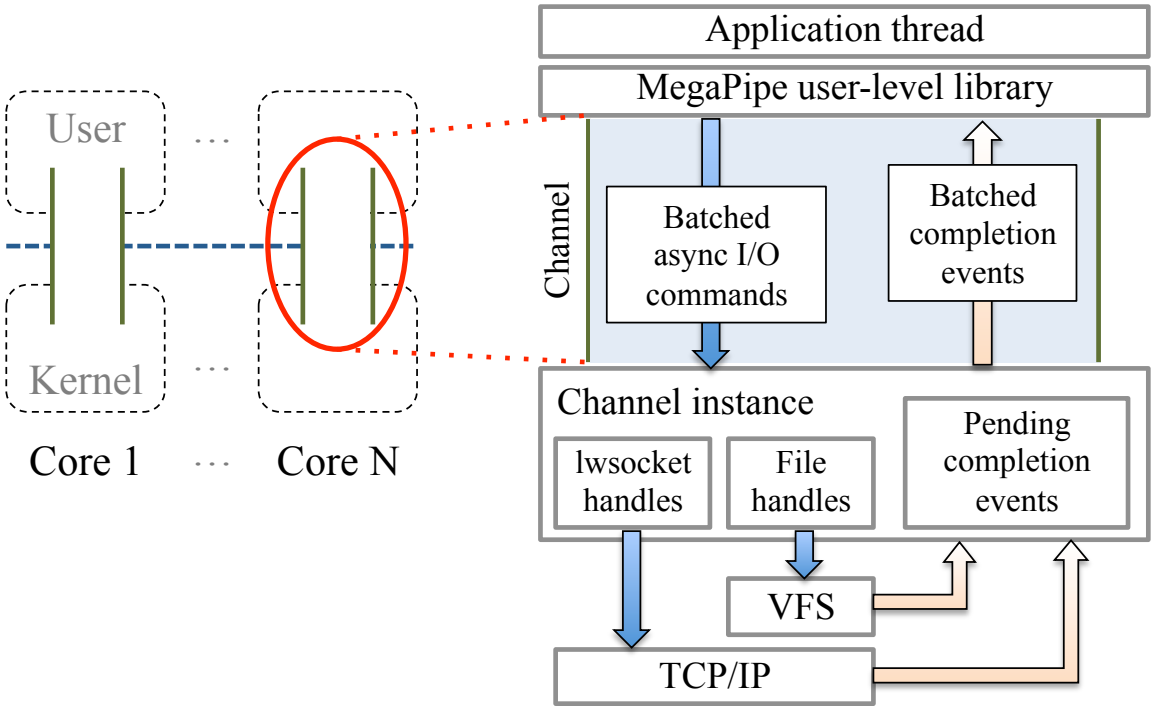


Figure 3.2: MegaPipe architecture

Each channel is composed of two message streams: a request stream and a completion stream. User-level applications issue asynchronous I/O requests to the kernel via the request stream. Once the asynchronous I/O request is done, the completion notification of the request is delivered to user-space via the completion stream. This process is done in a *batched* (§3.3.4) manner, to minimize the context switch between user and kernel. The MegaPipe user-level library is fully responsible for transparent batching; MegaPipe does not need to be aware of batching.

3.3.4 Design Components

Listening Socket Partitioning

As discussed in §3.2.1, the shared listening socket causes two issues in the multi-core context: *i)* contention on the accept queue and *ii)* cache bouncing between RX and TX cores for a flow. Affinity-Accept [91] proposes two key ideas to solve these issues. First, a listening socket has per-core accept queues instead of the shared one. Second, application threads that call `accept()` prioritize their local accept queue. In this way, connection establishment becomes completely parallelizable and independent, and all the connection establishment, data transfer, and application logic for a flow are contained in the same core.

In MegaPipe, we achieve essentially the same goals but with a more controlled approach. When an application thread associates a listening socket to a channel, MegaPipe spawns a

separate listening socket. The new listening socket has its own accept queue which is only responsible for connections established on a particular subset of cores that are explicitly specified by an optional `cpu_mask` parameter. After a shared listening socket is registered to MegaPipe channels with disjoint `cpu_mask` parameters, all channels (and thus cores) have completely partitioned backlog queues. Upon receipt of an incoming TCP handshaking packet, which is distributed across cores either by RSS [102] or RPS [45], the kernel finds a "local" accept queue among the partitioned set, whose `cpu_mask` includes the current core. On the application side, an application thread accepts pending connections from its local queue. In this way, cores no longer contend for the shared accept queue, and connection establishment is vertically partitioned (from the TCP/IP stack up to the application layer).

We briefly discuss the main difference between our technique and that of Affinity-Accept. Our technique requires user-level applications to partition a listening socket explicitly, rather than transparently. The downside is that legacy applications do not benefit. However, explicit partitioning provides more flexibility for user applications (e.g., to forgo partitioning for single-thread applications, to establish one accept queue for each physical core in SMT systems [127], etc.) Our approach follows the design philosophy of the Corey operating system, in a way that "applications should control sharing" [19].

Partitioning of a listening socket may cause potential load imbalance between cores [91]. Affinity-Accept handles two types of load imbalance. For short-term load imbalance, a non-busy core running `accept()` may steal a connection from the remote accept queue on a busy CPU core. For long-term load imbalance, the flow group migration mechanism lets the NIC to distribute more flows to non-busy cores. While the current implementation of MegaPipe does not support load balancing of incoming connections between cores, the techniques made in Affinity-Accept are complementary to MegaPipe. We leave the implementation and evaluation of connection load balancing as future work.

lwsocket: Lightweight Socket

`accept()`ing an established connection is an expensive process in the context of the VFS layer. In Unix-like operating systems, many different types of open files (disk files, sockets, pipes, devices, etc.) are identified by a *file descriptor*. A file descriptor is an integer identifier used as an indirect reference to an opened *file instance*, which maintains the status (e.g., access mode, offset, and flags such as `O_DIRECT` and `O_SYNC`) of the opened file. Multiple file instances may point to the same *inode*, which represents a unique, permanent file object. An inode points to an actual type-specific kernel object, such as TCB.

These layers of abstraction offer clear advantages. The kernel can seamlessly support various file systems and file types, while retaining a unified interface (e.g., `read()` and `write()`) to user-level applications. The CPU overhead that comes with the abstraction is tolerable for regular disk files, as file I/O is typically bound by low disk bandwidth or high seek latency. For network sockets, however, we claim that these layers of abstraction could be overkill for the following reasons:

1. *Sockets are rarely shared.* For disk files, it is common that multiple processes share the

same open file or independently open the same permanent file. The layer of indirection that file objects offer between the file table and inodes is useful in such cases. In contrast, since network sockets are rarely shared by multiple processes (HTTP socket redirected to a CGI process is such an exception) and not opened multiple times, this indirection is typically unnecessary.

2. *Sockets are ephemeral.* Unlike permanent disk-backed files, the lifetime of network sockets ends when they are closed. Every time a new connection is established or torn down, its FD, file instance, inode, and dentry are newly allocated and freed. In contrast to disk files whose inode and dentry objects are cached [64], socket inode and dentry cannot benefit from caching since sockets are ephemeral. The cost of frequent (de)allocation of those objects is exacerbated on multi-core systems since the kernel maintains the inode and dentry as globally visible data structures [20].

To address these issues, we propose lightweight sockets – *lwsocket*. Unlike regular files, a lwsocket is identified by an arbitrary integer within the channel, not the lowest possible integer within the process. The lwsocket is a common-case optimization for network connections; it does not create a corresponding file instance, inode, or dentry, but provides a straight shortcut to the TCB in the kernel. A lwsocket is only locally visible within the associated MegaPipe channel, which avoids global synchronization between cores.

In MegaPipe, applications can choose whether to fetch a new connection as a regular socket or as a lwsocket. Since a lwsocket is associated with a specific channel, one cannot use it with other channels or for general system calls, such as `sendmsg()`. In cases where applications need the full generality of file descriptors, MegaPipe provides a fall-back API function to convert a lwsocket into a regular file descriptor.

System Call Batching

Recent research efforts report that system calls are expensive not only due to the cost of mode switching, but also because of the negative effect on cache locality in both user and kernel space [119]. To amortize system call costs, MegaPipe batches multiple I/O requests and their completion notifications into a single system call. The key observation here is that batching can exploit connection-level parallelism, extracting multiple independent requests and notifications from concurrent connections.

Batching is transparently done by the MegaPipe user-level library for both directions user $\rightarrow$ kernel and kernel $\rightarrow$ user. Application programmers need not be aware of batching. Instead, application threads issue one request at a time, and the user-level library accumulates them. When *i)* the number of accumulated requests reaches the batching threshold, *ii)* there are not any more pending completion events from the kernel, or *iii)* the application explicitly asks to flush, then the collected requests are flushed to the kernel in a batch through the channel. Similarly, application threads dispatch a completion notification from the user-level library one by one. When the user-level library has no more completion notifications to feed the application thread, it fetches multiple pending notifications from kernel

Function	Parameters	Description
`mp_create()`		Create a new MegaPipe channel instance.
`mp_destroy()`	channel	Close a MegaPipe channel instance and clean up associated resources.
`mp_register()`	channel, fd, cookie, cpu_mask	Create a MegaPipe handle for the specified file descriptor (either regular or lightweight) in the given channel. If a given file descriptor is a listening socket, an optional CPU mask parameter can be used to designate the set of CPU cores which will respond to incoming connections for that handle.
`mp_unregister()`	handle	Remove the target handle from the channel. All pending completion notifications for the handle are canceled.
`mp_accept()`	handle, count, is_lwsocket	Accept one or more new connections from a given listening handle asynchronously. The application specifies whether to accept a connection as a regular socket or a lwsocket. The completion event will report a new FD/lwsocket and the number of pending connections in the accept queue.
`mp_read()` `mp_write()`	handle, buf, size	Issue an asynchronous I/O request. The completion event will report the number of bytes actually read/written.
`mp_writev()`	handle, iovec, iovcnt	Issue an asynchronous, vectored write with multiple buffers ("gather output")
`mp_close()`	handle	Close the connection (if still connected) and free up the handle.
`mp_disconnect()`	handle	Close the connection in a similar manner to `shutdown()`. It does not deallocate or unregister the handle.
`mp_close()`	handle	Close the connection if necessary, and free up the handle.
`mp_flush()`	channel	For the given channel, push any pending asynchronous I/O commands to the kernel immediately, rather than waiting for automatic flush events.
`mp_dispatch()`	channel, timeout	Poll for a single completion notification from the given channel. If there is no pending notification event, the call blocks until the specified timer expires. The timeout can be 0 (non-blocking) or -1 (indefinite blocking)

Table 3.1: List of MegaPipe API functions for userspace applications. All I/O functions are enqueued in the channel—instead of individually invoking a system call—and are deliviered to kernel in a batch.

in a batch. We set the default batching threshold to 32 (adjustable), as we found that the marginal performance gain beyond that point is negligible.

One potential concern is that batching may affect the latency of network servers since I/O requests are queued. However, batching happens only when the server is overloaded, so it does not affect latency when underloaded. Even if the server is overloaded, the additional latency should be minimal because the batch threshold is fairly small, compared to the large queues (thousands of packets) of modern NICs[50].

3.3.5 Application Programming Interface

The MegaPipe user-level library provides a set of API functions to hide the complexity of batching and the internal implementation details. Table 3.1 presents a partial list of MegaPipe API functions. Due to lack of space, we highlight some interesting aspects of some functions rather than enumerating all of them.

The application associates a handle (either a regular file descriptor or a lwsocket) with the specified channel with `mp_register()`. All further I/O commands and completion notifications for the registered handle are done through only the associated channel. A cookie, an opaque pointer for developer use, is also passed to the kernel with handle registration. This cookie is attached in the completion events for the handle, so the application can easily identify which handle fired each event. The application calls `mp_unregister()` to end the membership. Once unregistered, the application can continue to use the regular FD with general system calls. In contrast, lwsockets are immediately deallocated from the kernel memory.

When a listening TCP socket is registered with the `cpu_mask` parameter, MegaPipe internally spawns an accept queue for incoming connections on the specified set of CPU cores. The original listening socket (now responsible for the remaining CPU cores) can be registered to other MegaPipe channels with a disjoint set of cores – so each thread can have a completely partitioned view of the listening socket.

`mp_read()` and `mp_write()` issue asynchronous I/O commands. The application should not use the provided buffer for any other purpose until the completion event, as the ownership of the buffer has been delegated to the kernel, like in other asynchronous I/O APIs. The completion notification is fired when the I/O is actually completed, i.e., all data has been copied from the receive queue for read or copied to the send queue for write. In adapting nginx and memcached, we found that vectored I/O operations (multiple buffers for a single I/O operation) are helpful for optimal performance. For example, the unmodified version of nginx invokes the `writev()` system call to transmit separate buffers for a HTTP header and body at once. MegaPipe supports the counterpart, `mp_writev()`, to avoid issuing multiple `mp_write()` calls or aggregating scattered buffers into one contiguous buffer.

`mp_dispatch()` returns one completion event as a `struct mp_event`. This data structure contains: *i)* a completed command type (e.g., read/write/accept/etc.), *ii)* a cookie, *iii)* a result field that indicates success or failure (such as broken pipe or connection reset) with the corresponding `errno` value, and *iv)* a `union` of command-specific return values.

```
ch = mp_create()
handle = mp_register(ch, listen_sd, mask=0x01)
mp_accept(handle)

while true:
  ev = mp_dispatch(ch)
  conn = ev.cookie
  if ev.cmd == ACCEPT:
    mp_accept(conn.handle)
    conn = new Connection()
    conn.handle = mp_register(ch, ev.fd, cookie=conn)
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == READ:
    mp_write(conn.handle, conn.buf, ev.size)
  elif ev.cmd == WRITE:
    mp_read(conn.handle, conn.buf, READSIZE)
  elif ev.cmd == DISCONNECT:
    mp_unregister(ch, conn.handle)
    delete conn
```

Listing 3.1: Pseudocode for ping-pong server event loop

Listing 3.1 presents simplified pseudocode of a ping-pong server to illustrate how applications use MegaPipe. An application thread initially creates a MegaPipe channel and registers a listening socket (`listen_sd` in this example) with `cpu_mask` 0x01 (first bit is set) which means that the handle is only interested in new connections established on the first core (core 0). The application then invokes `mp_accept()` and is ready to accept new connections. The body of the event loop is fairly simple; given an event, the server performs any appropriate tasks (barely anything in this ping-pong example) and then fires new I/O operations.

Currently MegaPipe provides only I/O-related functionality. We note that this asynchronous communication between kernel and user applications can be applied to other types of system calls as well. For example, we could invoke the `futex()` system call, which is not related to file operations, to grab a lock asynchronously. One natural approach to support general system calls in MegaPipe would be having a new API function `mp_syscall(syscall number, arg1, ...)`.

3.3.6 Discussion: Thread-Based Servers

The MegaPipe design naturally fits event-driven servers based on callback or event-loop mechanisms, for example, Flash [87] and SEDA [132]. We mostly focus on event-driven servers in this work. On the other hand, MegaPipe is also applicable to thread-based servers,

by having one channel for each thread, thus each connection. In this case the application cannot take advantage of batching (§3.3.4), since batching exploits the parallelism of independent connections that are multiplexed through a channel. However, the application still can benefit from partitioning (§3.3.4) and lwsocket (§3.3.4) for better scalability on multi-core servers.

There is an interesting spectrum between pure event-driven servers and pure thread-based servers. Some frameworks expose thread-like environments to user applications to retain the advantages of thread-based architectures, while looking like event-driven servers to the kernel to avoid the overhead of threading. Such functionality is implemented in various ways: lightweight user-level threading [13, 29], closures or coroutines [17, 67, 85], and language runtime [7]. Those frameworks intercept I/O calls issued by user threads to keep the kernel thread from blocking, and manage the outstanding I/O requests with polling mechanisms, such as `epoll`. These frameworks can leverage MegaPipe for higher network I/O performance without requiring modifications to applications themselves. We leave the evaluation of effectiveness of MegaPipe for these frameworks as future work.

3.4 Implementation

This section describes how we implemented MegaPipe in the Linux kernel and the associated user-level library. As briefly described in §3.3.3, MegaPipe consists of two parts: the kernel module and the user-level library. In this section, we denote them by MP-K and MP-L, respectively, for clear distinction between the two.

3.4.1 Kernel Implementation

MP-K interacts with MP-L through a special device, `/dev/megapipe`. MP-L opens this file to create a channel, and invokes `ioctl()` system calls on the file to issue I/O requests and dispatch completion notifications for that channel.

MP-K maintains a set of handles for both regular FDs and lwsockets in a red-black tree[3] for each channel. Unlike a per-process file table, each channel is only accessed by one thread, avoiding data sharing between threads (thus cores). MP-K identifies a handle by an integer unique to the owning channel. For regular FDs, the existing integer value is used as an identifier, but for lwsockets, an integer of 2^{30} or higher value is issued to distinguish lwsockets from regular FDs. This range is used since it is unlikely to conflict with regular FD numbers, as the POSIX standard allocates the lowest unused integer for FDs[124].

MP-K currently supports the following file types: sockets, pipes, FIFOs, signals (via `signalfd`), and timers (via `timerfd`). MP-K handles asynchronous I/O requests differently depending on the file type. For sockets (such as TCP, UDP, and UNIX domain), MegaPipe utilizes the native callback interface, which fires upon state changes, supported

[3] It was mainly for ease of implementation, as Linux provides the template of red-black trees. We have not yet evaluated alternatives, such as a hash table, which supports $O(1)$ lookup rather than $O(logN)$.

by kernel sockets for optimal performance. For other file types, MP-K internally emulates asynchronous I/O with `epoll` and non-blocking VFS operations within kernel. MP-K currently does not support disk files, since the Linux file system does not natively support asynchronous or non-blocking disk I/O, unlike other modern operating systems. One approach to work around this issue is to adopt a lightweight technique presented in FlexSC [119] to emulate asynchronous I/O. When a disk I/O operation is about to block, MP-K can spawn a new thread on demand while the current thread continues.

Upon receiving batched I/O commands from MP-L through a channel, MP-K first examines if each request can be processed immediately (e.g., there is pending data in the TCP receive queue, or there is free space in the TCP send queue). If so, MP-K processes the request and issues a completion notification immediately, without incurring the callback registration or `epoll` overhead. This idea of opportunistic shortcut is adopted from LAIO [27], where the authors claim that the 73–86% of I/O operations are readily available. For I/O commands that are not readily available, MP-K needs some bookkeeping; it registers a callback to the socket or declares an epoll interest for other file types. When MP-K is notified that the I/O operation has become ready, it processes the operation.

MP-K enqueues I/O completion notifications in the per-channel event queue. Those notifications are dispatched in a batch upon the request of MP-L. Each handle maintains a linked list to its pending notification events, so that they can be easily removed when the handle is unregistered (and thus not of interest anymore).

We implemented MP-K in the Linux 3.1.3 kernel with 2,200 lines of code in total. The majority was implemented as a Linux kernel module, such that the module can be used for other Linux kernel versions as well. However, we did have to make three minor modifications (about 400 lines of code of the 2,200) to the Linux kernel itself, due to the following issues: *(i)* we modified `epoll` to expose its API to not only user space but also to MP-K; *(ii)* we modified the Linux kernel to allow multiple sockets (partitioned) to listen on the same address/port concurrently, which traditionally is not allowed; and *(iii)* we also enhanced the socket lookup process for incoming TCP handshake packets to consider `cpu_mask` when choosing a destination listening socket among a partitioned set.

3.4.2 User-Level Library

MP-L is essentially a simple wrapper of the kernel module, and it is written in about 400 lines of code. MP-L performs two main roles: *i)* it transparently provides batching for asynchronous I/O requests and their completion notifications, *ii)* it performs communication with MP-K via the `ioctl()` system call.

The current implementation uses copying to transfer commands (24 B for each) and notifications (40 B for each) between MP-L and MP-K. This copy overhead, roughly 3–5% of total CPU cycles (depending on workloads) in our evaluation, can be eliminated with virtual memory mapping for the command/notification queues, as introduced in Mach Port [2]. We leave the implementation and evaluation of this idea as future work.

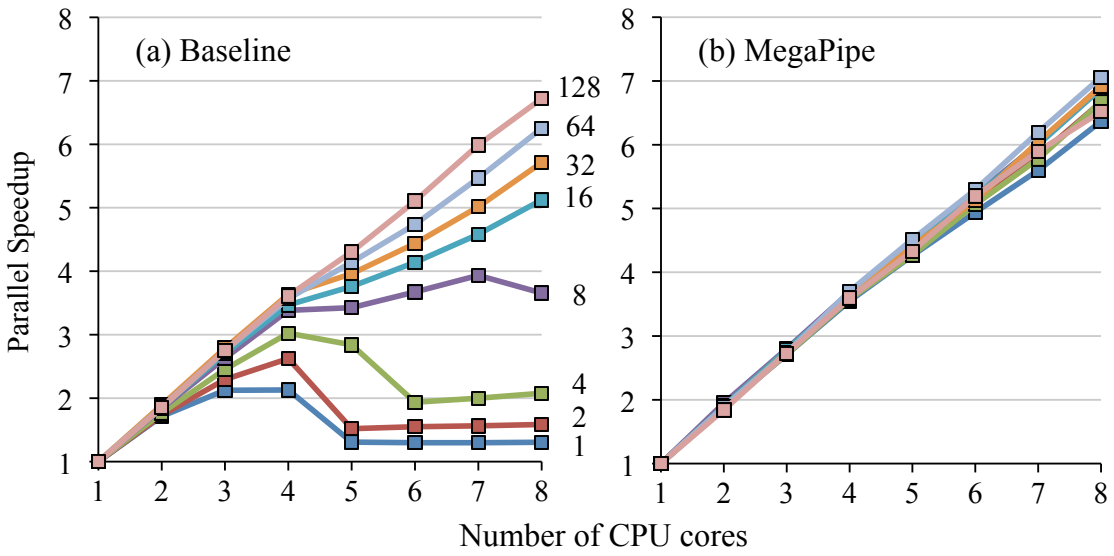


Figure 3.3: Comparison of parallel speedup for varying numbers of transactions per connection (labeled) over a range of CPU cores (x-axis) with 64 B messages.

3.5 Evaluation

We evaluated the performance gains yielded by MegaPipe both through a collection of microbenchmarks, akin to those presented in §3.2.2. Unless otherwise noted, all benchmarks were completed with the same experimental setup (same software versions and hardware platforms as described in §3.2.2.

The purpose of the microbenchmark results is three-fold. First, utilization of the same benchmark strategy as in §3.2 allows for direct evaluation of the low-level limitations we previously highlighted. Figure 3.1 shows the performance of MegaPipe measured for the same experiments. Second, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as the minimal benchmark program effectively rules out any complex effects from application-specific behaviors. Third, microbenchmarks allow us to illuminate the performance contributions of each of MegaPipe's individual design components.

3.5.1 Multi-Core Scalability

We begin with the impact of MegaPipe on multi-core scalability. Figure 3.3 provides a side-by-side comparison of parallel speedup (compared to the single core case of each) for a variety of transaction lengths. The baseline case on the left clearly shows that the scalability highly depends on the length of connections. For short connections, the throughput stagnates as core count grows due to the serialization at the shared accept queue, then suddenly collapses with more cores. We attribute the performance collapse to increased cache congestion and non-scalable locks [21]; note that the connection establishment process happens more frequently with short flows in our test, increasing the level of contention.

	Number of transactions per connection							
	1	2	4	8	16	32	64	128
+P	211.6	207.5	181.3	83.5	38.9	29.5	17.2	8.8
P +B	18.8	22.8	72.4	44.6	31.8	30.4	27.3	19.8
PB +L	352.1	230.5	79.3	22.0	9.7	2.9	0.4	0.1
Total	582.4	460.8	333.1	150.1	80.4	62.8	45.0	28.7

Table 3.2: Accumulation of throughput improvement (%) over baseline, from three contributions of MegaPipe.

In contrast, the throughput of MegaPipe scales almost linearly regardless of connection length, showing speedup of 6.4 (for single-transaction connections) or higher. This improved scaling behavior of MegaPipe is mostly from the multi-core related optimizations techniques, namely partitioning and lwsocket. We observed similar speedup without batching, which enhances per-core throughput.

3.5.2 Breakdown of Performance Improvement

In Table 3.2, we present the incremental improvements (in percent over baseline) that Partitioning (P), Batching (B), and lwsocket (L) contribute to overall throughput, by accumulating each technique in that order. In this experiment, we used all eight cores, with 64 B messages (1 KiB messages yielded similar results). Both partitioning and lwsocket significantly improve the throughput of short connections, and their performance gain diminishes for longer connections since the both techniques act only at the connection establishment stage. For longer connections (not shown in the table), the gain from batching converged around 15%. Note that the case with partitioning alone (+P in the table) can be seen as *sockets with Affinity-Accept* [91], as the both address the shared accept queue and connection affinity issues. lwsocket further contributes the performance of short connections, helping to achieve near-linear scalability as shown in Figure 3.3(b).

3.5.3 Impact of Message Size

Lastly, we examine how the improvement changes by varying message sizes. Figure 3.4 depicts the relative throughput improvement, measured with 10-transaction connections. For the single-core case, where the improvement comes mostly from batching, MegaPipe outperforms the baseline case by 15–33%, showing higher effectiveness for small ($\leq$ 1 KiB) messages. The improvement goes higher as we have five or more cores, since the baseline case experiences more expensive off-chip cache and remote memory access, while MegaPipe effectively mitigates them with partitioning and lwsocket. The degradation of relative improvement from large messages with many cores reflects that the server was able to saturate the 10 G link. MegaPipe saturated the link with seven, five, and three cores for 1, 2, and

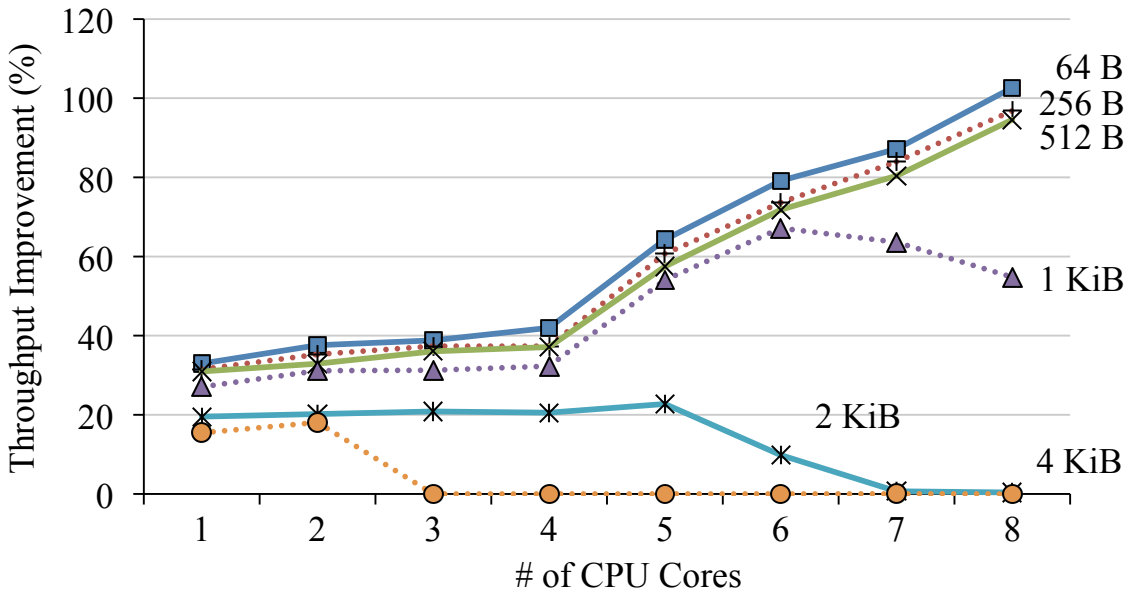


Figure 3.4: Relative performance improvement for varying message sizes over a range of CPU cores.

4 KiB messages, respectively. The baseline Linux saturated the link with seven and three cores for 2 and 4 KiB messages, respectively.

3.6 Conclusion

Message-oriented network workloads, where connections are short and/or message sizes are small, are CPU-intensive and scale poorly on multi-core systems with the BSD Socket API. In this chapter, we introduced MegaPipe, a new programming interface for high-performance networking I/O. MegaPipe exploits many performance optimization opportunities that were previously hindered by existing network API semantics, while being still simple and applicable to existing event-driven servers with moderate efforts. Evaluation through microbenchmarks, memcached, and nginx showed significant improvements, in terms of both single-core performance and parallel speedup on an eight-core system.

Chapter 4

Applications of MegaPipe

4.1 Adopting Existing Server Applications

To verify the applicability of MegaPipe, we adapted two popular event-driven servers, memcached 1.4.13 [77] (an in-memory key-value store) and nginx 1.0.15 [123] (a web server), to verify the applicability of MegaPipe. As quantitatively indicated in Table 4.1, the code changes required to use MegaPipe were manageable, on the order of hundreds of lines of code. However, these two applications presented different levels of effort during the adaptation process. We briefly introduce our experiences here, and show the performance benefits in Section 3.5.

Application	Total	Changed
memcached	9,442	602 (6.4%)
nginx	86,774	447 (0.5%)

Table 4.1: Lines of code for application adaptations

4.1.1 Porting memcached

memcached uses the libevent [75] framework which is based on the readiness model (e.g., `epoll` on Linux). The server consists of a main thread and a collection of worker threads. The main thread accepts new client connections and distributes them among the worker threads. The worker threads run event loops which dispatch events for client connections.

Modifying memcached to use MegaPipe in place of libevent involved three steps:

1. **Decoupling from libevent**: We began by removing libevent-specific data structures from memcached. We also made the drop-in replacement of `mp_dispatch()` for the libevent event dispatch loop.

2. **Parallelizing accept**: Rather than having a single thread that accepts all new connections, we modified worker threads to accept connections in parallel by partitioning the shared listening socket.

3. **State machine adjustment**: Finally, we replaced calls to `read()` with `mp_read()` and calls to `sendmsg()` with `mp_writev()`. Due to the semantic gap between the readiness model and the completion notification model, each state of the memcached state machine that invokes a MegaPipe function was split into two states: actions prior to a MegaPipe function call, and actions that follow the MegaPipe function call and depend on its result. We believe this additional overhead could be eliminated if memcached did not have the strong assumption of the readiness model.

In addition, we pinned each worker thread to a CPU core for the MegaPipe adaptation, which is considered a best practice and is necessary for MegaPipe. We made the same modification to stock memcached for a fair comparison.

4.1.2 Porting nginx

Compared to memcached, nginx modifications were much more straightforward due to three reasons: *i)* the custom event-driven I/O of nginx does not use an external I/O framework that has a strong assumption of the readiness model, such as libevent [75]; *ii)* nginx was designed to support not only the readiness model (by default with `epoll` in Linux), but also the completion notification model (for POSIX AIO [124] and signal-based AIO), which nicely fits with MegaPipe; and *iii)* all worker processes already accept new connections in parallel, but from the shared listening socket.

nginx has an extensible *event module* architecture, which enables easy replacement for its underlying event-driven mechanisms. Under this architecture, we implemented a MegaPipe event module and registered `mp_read()` and `mp_writev()` as the actual I/O functions. We also adapted the worker threads to accept new connections from the partitioned listening socket.

4.2 Macrobenchmark: memcached

We perform application-level macrobenchmarks of memcached, comparing the baseline performance to that of memcached adapted for MegaPipe as previously described. For baseline measurements, we used a patched[1] version of the stock memcached 1.4.13 release.

We used the `memaslap` [3] tool from libmemcached 1.0.6 to perform the benchmarks. We patched `memaslap` to accept a parameter designating the maximum number of requests to issue for a given TCP connection (upon which it closes the connection and reconnects to the server). Note that the typical usage of memcached is to use persistent connections to servers

[1] We discovered a performance bug in the stock memcached release as a consequence of unfairness towards servicing new connections, and we corrected this fairness bug.

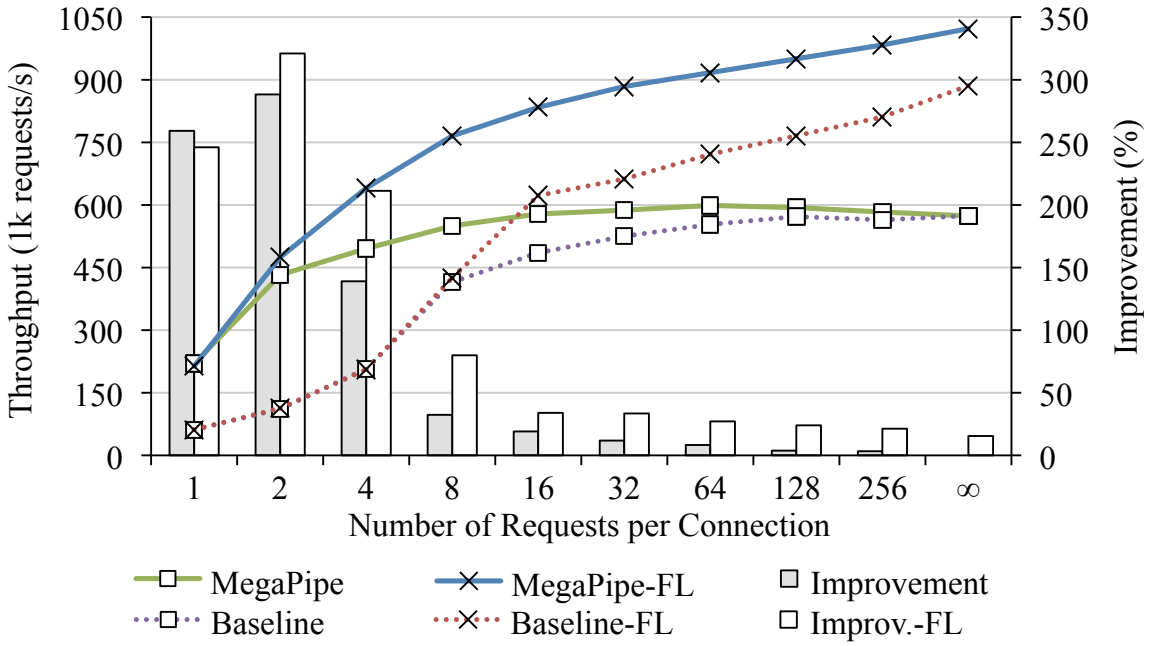


Figure 4.1: memcached throughput comparison with eight cores, by varying the number of requests per connection. ∞ indicates persistent connections. Lines with "X" markers (-FL) represent fine-grained-lock-only versions.

or UDP sockets, so the performance result from short connections may not be representative of memcached; rather, it should be interpreted as what-if scenarios for event-driven server applications with non-persistent connections.

The key-value workload used during our tests is the default `memaslap` workload: 64 B keys, 1 KiB values, and a get/set ratio of 9:1. For these benchmarks, each of three client machines established 256 concurrent connections to the server. On the server side, we set the memory size to 4 GiB. We also set the initial hash table size to 2^{22} (enough for 4 GiB memory with 1 KiB objects), so that the server would not exhibit performance fluctuations due to dynamic hash table expansion during the experiments.

Figure 4.1 compares the throughput between the baseline and MegaPipe versions of memcached (we discuss the "-FL" versions below), measured with all eight cores. We can see that MegaPipe greatly improves the throughput for short connections, mostly due to partitioning and lwsocket as we confirmed with the microbenchmark. However, the improvement quickly diminishes for longer connections, and for persistent connections, MegaPipe does not improve the throughput at all. Since the MegaPipe case shows about 16% higher throughput for the single-core case (not shown in the graph), it is clear that there is a performance-limiting bottleneck for the multi-core case. Profiling reveals that spin-lock contention takes roughly 50% of CPU cycles of the eight cores, highly limiting the scalability.

In memcached, normal get/set operations involve two locks: `item_locks` and a global lock `cache_lock`. The fine-grained `item_locks` (the number is dynamic, 8,192 locks on eight

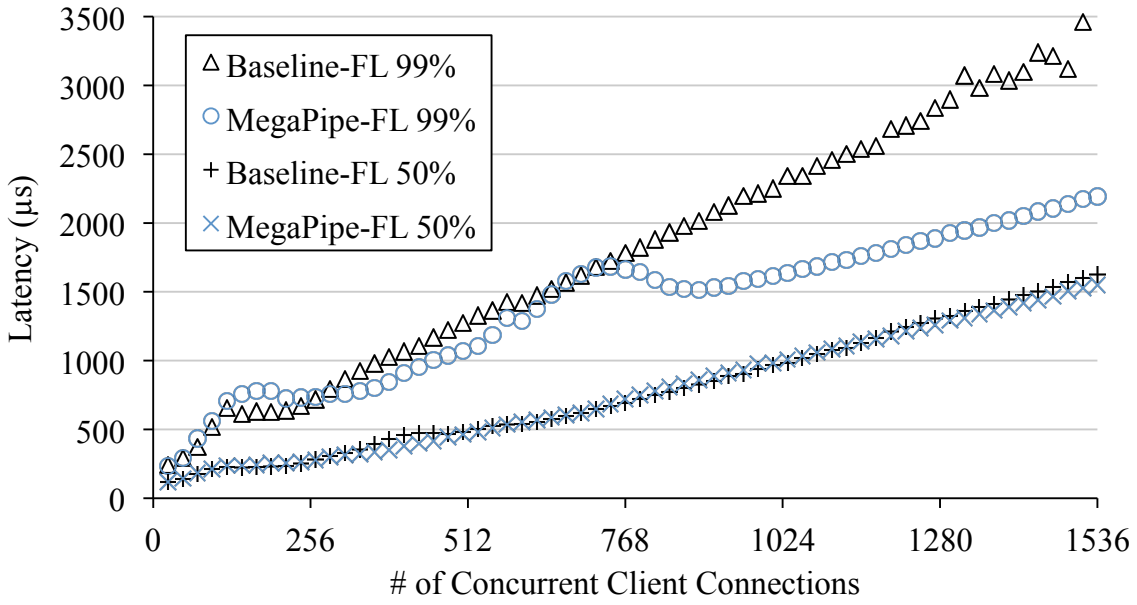


Figure 4.2: 50th and 99th percentile memcached latency.

cores) keep the consistency of the object store from concurrent accesses by worker threads. On the other hand, the global `cache_lock` ensures that the hash table expansion process by the *maintenance thread* does not interfere with worker threads. While this global lock is inherently not scalable, it is unnecessary for our experiments since we configured the hash table expansion to not happen by giving a sufficiently large initial size.

We conducted experiments to see what would happen if we rule out the global lock, thus relying on the fine-grained locks (`item_locks`) only. We provide the results (with the suffix "-FL") also in Figure 4.1. Without the global lock, the both MegaPipe and baseline cases perform much better for long or persistent connections. For the persistent connection case, batching improved the throughput by 15% (note that only batching among techniques in §5.2 affects the performance of persistent connections). We can conclude two things from these experiments. First, MegaPipe improves the throughput of applications with short flows, and the improvement is fairly insensitive to the scalability of applications themselves. Second, MegaPipe might not be effective for poorly scalable applications, especially with long connections.

Lastly, we discuss how MegaPipe affects the latency of memcached. One potential concern with latency is that MegaPipe may add additional delay due to batching of I/O commands and notification events. To study the impact of MegaPipe on latency, we measured median and tail (99th percentile) latency observed by the clients, with varying numbers of persistent connections, and plotted these results in Figure 4.2. The results show that MegaPipe does not adversely affect the median latency. Interestingly, for the tail latency, MegaPipe slightly increases it with low concurrency (between 72–264) but greatly reduces it with high concurrency ($\geq$ 768). We do not fully understand these tail behaviors yet. One likely explanation

for the latter is that batching becomes more effective with high concurrency; since that batching exploits parallelism from independent connections, high concurrency yields larger batch sizes.

We conduct all experiments with the interrupt coalescing feature of the NIC. We briefly describe the impact of disabling it, to investigate if MegaPipe favorably or adversely interfere with interrupt coalescing. When disabled, the server yielded up to 50μs (median) and 200μs (tail) lower latency with low concurrency (thus underloaded). On the other hand, beyond near saturation point, disabling interrupt coalescing incurred significantly higher latency due to about 30% maximum throughput degradation, which causes high queueing delay. We observed these behaviors for both MegaPipe and baseline; we could not find any MegaPipe-specific behavior with interrupt coalescing in our experiments.

4.3 Macrobenchmark: nginx

Unlike memcached, the architecture of nginx is highly scalable on multi-core servers. Each worker process has an independent address space, and nothing is shared by the workers, so the performance-critical path is completely lockless. The only potential factor that limits scalability is the interface between the kernel and user, and we examine how MegaPipe improves the performance of nginx with such characteristics.

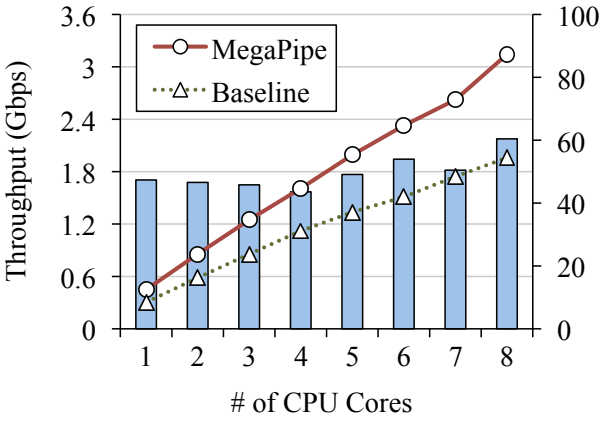
For the nginx HTTP benchmark, we conduct experiments with three workloads with static content, namely `SpecWeb`, `Yahoo`, and `Yahoo/2`. For all workloads, we configured nginx to serve files from memory rather than disks, to avoid disks being a bottleneck. We used weighttp[2] as a workload generator, and we modified it to support variable number of requests per connection.

1. **SpecWeb**: We test the same HTTP workload used in Affinity-Accept [91]. In this workload, each client connection initiates six HTTP requests. The content size ranges from 30 to 5,670 B (704 B on average), which is adopted from the static file set of SpecWeb 2009 Support Workload [121].

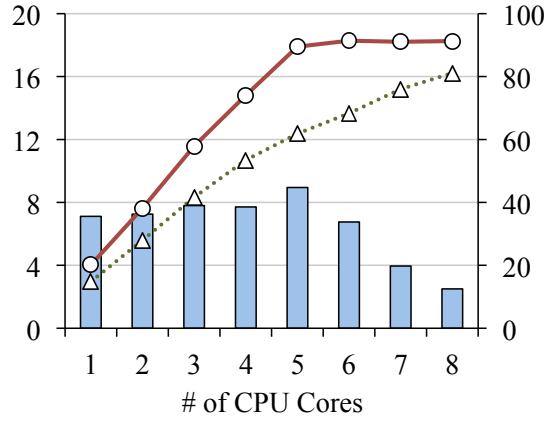
2. **Yahoo**: We used the HTTP trace collected from the Yahoo! CDN [32]. In this workload, the number of HTTP requests per connection ranges between 1 and 1,597. The distribution is heavily skewed towards short connections (98% of connections have ten or less requests, 2.3 on average), following the Zipf-like distribution. Content sizes range between 1 B and 253 MiB (12.5 KiB on average). HTTP responses larger than 60 KiB contribute roughly 50% of the total traffic.

3. **Yahoo/2**: Due to the large object size of the `Yahoo` workload, MegaPipe with only five cores saturates the two 10G links we used. For the `Yahoo/2` workload, we change the size of all files by half, to avoid the link bottleneck and observe the multi-core scalability behavior more clearly.

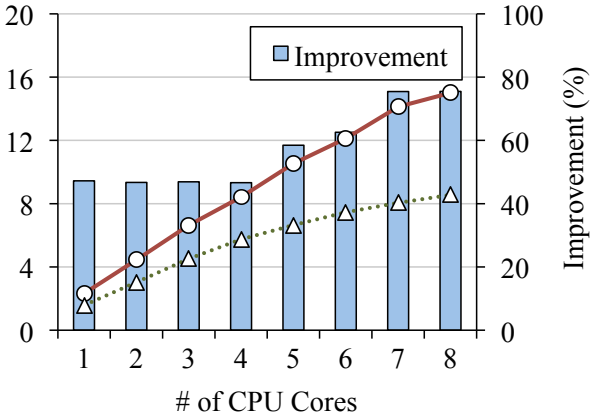
[2] http://redmine.lighttpd.net/projects/weighttp/wiki



(a) SpecWeb



(b) Yahoo



(c) Yahoo/2

Figure 4.3: Evaluation of nginx throughput for various workloads.

Web servers can be seen as one of the most promising applications of MegaPipe, since typical HTTP connections are short and carry small messages [32]. We present the measurement result in Figure 4.3 for each workload. For all three workloads, MegaPipe significantly improves the performance of both single-core and multi-core cases. MegaPipe with the `Yahoo/2` workload, for instance, improves the performance by 47% (single core) and 75% (eight cores), with a better parallel speedup (from 5.4 to 6.5) with eight cores. The small difference of improvement between the `Yahoo` and `Yahoo/2` cases, both of which have the same connection length, shows that MegaPipe is more beneficial with small message sizes.

Chapter 5

BESS: A Modular Framework for Extensible Network Dataplane

One of the prominent trends in computer networking is the increasing role of software in network dataplane. In the previous chapter we discussed the traditional, predominant use: socket-level server applications running on network endpoints. In addition to this, there are a growing number of packet-level in-network applications, in the form of network dataplane software running on commercial-off-the-shelf (COTS) servers. As compared to custom solutions based on specialized hardware appliances powered by specialized ASIC and FGPA, software running on general-purpose hardware provides clear advantages. Software-based dataplane is more accessible (less prone to vendor lock-in) and flexible (easier to embrace experimental features and future changes).

These accessibility and flexibility of software dataplane had been outweighed by its seemingly inherent disadvantage compared to hardware solutions; software is too slow to meet the performance requirements of today's network workloads. Software was simply not a practical alternative for processing millions of packets per second at the microsecond timescale. What should give us pause, however, is the mounting evidence that the performance gap between hardware and software network dataplane is significantly getting smaller than previously believed. Recent progress in software network dataplane has been largely driven by two enablers: architectural improvements in general-purpose server on hardware [**pcie**, 24, 25] and the advent of kernel-bypassing packet I/O on software [41, 48, 103].

In this chapter we introduce BESS, short for Berkeley Extensible Software Switch, which is a programmable platform for network dataplane implementation. We designed BESS with the following three packet-oriented dataplane applications in mind, as its major use cases:

- NIC augmentation: this application refers to software implementation of advanced packet processing features that used to be implemented in the ASIC of network interface cards (NICs). Such NIC features include packet scheduling, segmentation/reassembly, classificaton, hardware slicing, etc. These features essentially "offload" the burden of the OS network stack, which is known to be notoriously slow and inefficient.

Recently, however, more and more hardware NIC features are now being implemented back in software, due to the various reasons as discussed in the following section.

- Virtual switching: virtual switching is a crucial component to implement server virtualization. Virtual switches are typically implemented at the hypervisor, providing network access to virtual machines (VMs). In addition to the basic connectivity, virtual switches are also responsible for additional features, for example, access control, migration support, tunneling, etc.

- Network function implementation: network function (NF) is an in-network entity performing packet processing. For example, switches and routers are network functions that provide network connectivity for end hosts. Network functions also include network "middleboxes", which provides more advanced functionality—or *services*—beyond simple packet forwarding [113], including firewall, proxy, virtual private network, etc., to name a few. While most network services have been implemented as proprietary physical appliances, the recent advent of network function virtualization (NFV) advocates NFs implemented in software, running on fully virtualized infrastructure.

While these applications differ from each other in terms of context and motivation, they still share very similar design goals and challenges. For instance, all of them need to implement a mechanism to provide some level of fairness (or inversely, differentiation) and performance isolation among "entities", whose meaning may depend on the applcation domains, e.g., i) individual TCP/UDP connections for NIC augmentation, ii) VMs for virtual switching, and iii) service subscribers for network functions. BESS provides a programmable platform for various types of network dataplane implementations. With BESS, dataplane developers can focus on the application-specific logic, rather than merely reinventing the wheel to implement common features with a large amount of boiler-plate code.

The BESS project initially focused on the particular appliation: software augmentation of NIC hardware, the first in the above list, hence with its former name, SoftNIC. During the development of our NFV system, E2 [88], we realized that BESS can be useful as a general-purpose dataplane framework beyond NIC augmentation. We have focused most of our BESS development efforts on supporting various dataplane features desired for E2, including: load balancing across virtual NF instances, tracking affinity between flows and instances, classifying packets for service chaining, etc., all in a distributed fashion. Since it is unlikely that such mechanisms are readily available in any existing vswitches, rather what is more important for us was the ability to accommodate new features easily and quickly. In addition to extensibility, performance was another indispensable goal for us, given that NFV systems should be able to handle large traffic aggregates within a sub-millisecond latency budget. Note that each packet may require multiple—even up to tens of—traverses over vswitches; the per-packet processing overhead of a vswitch must be minimal.

Besides BESS, we examined several existing vswitch implementations, and unfortunately none of them did not meet the both extensibility and performance requirements. Extending the Linux network stack was our first consideration, due to its maturity and universal

availability. However, its monolithic design forced a new feature to be tightly coupled to the rest of network stack, or even non-network subsystems in Linux kernel. Its infamous low performance—typically an order or magnitude slower than recent "kernel-bypass" solutions—was another deal breaker. There are a number of alternative vswitch implementations [94, 104, 130], but their forwarding logic is hard-coded with a limited means of programmability. For example, the datapath of Open vSwitch [94] is based on the OpenFlow match-action tables – any features that deviate from the semantics (e.g., "temporarily buffer out-of-order TCP packets for a short time") would require heavy modifications to its software architecture (i.e., tracking and arithmetic operations on TCP connection state, for the given example).

BESS is different from other existing software-based network functions in that it aims to provide a programmable framework, rather than an implementation of a specific forwarding logic. The datapath of BESS is composed in a declarative manner, as a data-flow graph of modules. Packets "flow" over the graph, and each module along the path performs module-specific operations, such as packet classification, filtering, encapsulation, and so on. An external controller can (incrementally) update the graph and configure individual modules, to construct a custom dataplane functionality.

This data-flow approach of BESS is heavily inspired by the Click modular router [65], while we both extend and simplify its design and implementation choices (§5.2). In particular, this paper details two major updates to the classic Click design. First, BESS introduces dynamic per-packet metedata attributes, which replace the static, inefficient, and error-prone notion of "annotations" in Click (§5.3). This feature eliminates the implicit compile-time dependency between modules, thus allowing better modularity. Second, BESS incorporates a resource-aware scheduler, which enables flexible scheduling policies (§5.4). The scheduler delivers fine-grained control over shared resources, such as processor time and link bandwidth, across tenants, VMs, applications, or flows.

5.1 Motivation

5.1.1 Software-Augmented Network Interface Card

The list of features that need to be supported by NICs has grown rapidly: new applications appear with new requirements; new protocols emerge and evolve; and there is no shortage of new ideas on system/network architectures that require new NIC features (*e.g.,* [4, 55, 82, 93, 96, 100]). Unfortunately, NIC hardware is not a silver bullet, as many issues arise due to the inherent inflexibility of hardware.

1. NICs may not support all desired features. Due to the slow development cycle (typically years), there often exists a *demand-availability gap*—a disparity between user demands for NIC features and their actual availability in products. For example, even minor updates (*e.g.,* do not duplicate the TCP CWR flag across all segments when using TSO, to avoid interference with congestion control algorithms [22, 101]) that can be

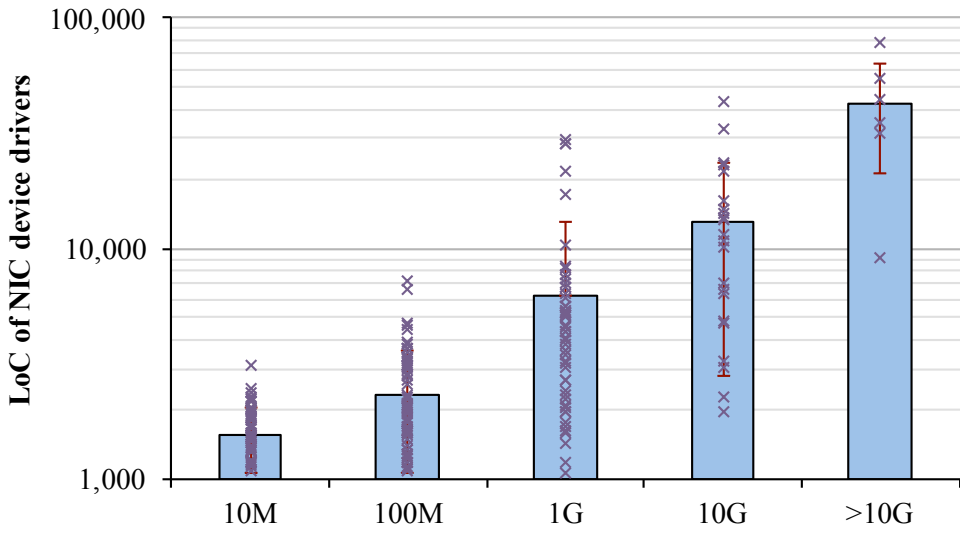


Figure 5.1: Growing complexity of NIC hardware, indirectly measured with the lines of device driver code in Linux 3.19.

implemented by changing a few lines of code when implemented in software, often requires a full hardware revision.

2. Once a feature is hardwired in the NIC, it is almost impossible to control its fine-grained behavior, often rendering the feature useless. For example, in the case of protocol offloading (*e.g.,* checksum offload and TSO/LRO), tunneled packets (*e.g.,* VXLAN) cannot take advantage of unless the NIC understands the encapsulation format [66], even though hardware essentially has the logic for protocol offloading built-in. Lack of fine-grain control also makes it difficult to combine features to achieve the end goal. For example, although SR-IOV provides better performance and isolation with hypervisor bypass, it cannot be used by cloud providers unless it supports additional features to enforce traffic policies such as security/QoS rules, advanced switching, and VM mobility support [96].

3. NIC hardware has resource restrictions, *e.g.,* the number of rate limiters, flow table size, and packet buffers, limiting its applicability. As a result, several systems [42, 60, 69, 92] have resorted to software work around the resource issue.

We argue that these issues will persist or even worsen in the future. Figure 5.1 presents the number of lines of code for NIC device drivers as an indirect index of NIC hardware complexity. The trends over the NIC generations show that the complexity has grown tremendously; *e.g.,* the driver for a high-end NIC (> 10 GbE) on average contains 6.8× more code than that of a 1 GbE NIC driver. This ever increasing hardware complexity has led to an increase in the time and cost for designing and verifying new NIC hardware.

BESS presents a new approach to extending NIC functionality; its adds a software shim layer between the NIC hardware and the network stack, so it can augment NIC features with software. It enables high-performance packet processing in software, while taking advantage of hardware features. By design, it supports high performance, extensibility, modular design, and backwards compatibility with existing software.

5.1.2 Hypervisor Virtual Switch

Virtual switch, or vswitch, is a key software component in datacenter computing. It provides network connectivity to virtualized compute resources, acting as the main network gateway to a physical host. The traditional role of vswitch was mere packet forwarding; i.e., simple bridging for VMs to the physical network. Today vswitches now host advanced features, such as monitoring, access control, traffic shaping, and network virtualization to name a few. The control/management plane of vswitch takes high-level network policies regarding such features and enforce them by configuring the data plane on individual hosts. From the data plane of vswitch, which is the main focus of this paper, we want two desirable, but often conflicting properties: *extensibility* and *performance*.

The need for extensibility comes from the fact that not only the use cases of vswitch, but also their environments and requirements keep diverging: e.g., multi-tenant datacenters, hybrid clouds, network function virtualization (NFV), and container-based virtualization systems. The environments and requirements of such use cases typically require various features in the data plane: new protocols, addressing and forwarding schemes, packet scheduling mechanisms, etc. Unfortunately—unlike the common assumptions—being written in software does not necessarily grant rapid updates with new functionality [112]. Even for open-source vswitches, in order to add a small feature, one has to deal with large and complex existing codebase, convince "upstream" maintainers of its necessity and usefulness, and wait for a new release and its wide-spread deployment. This whole process may take several months, if not years.

On the other hand, we want vswitch to process high bandwidth traffic at low latency with minimal CPU usage, so that the remaining compute resource can be used for the actual application logic. Unfortunately, the data plane performance tends to degrade as a vswitch gets sophisticated and incorporates more features, even if only a few of them are actually used. Performance optimization becomes difficult due to increased code/memory footprint and inter-dependency among features. Some systems avoid the "feature-bloat" issue with specialized and streamlined datapath implementations for specific vswitch use cases. While such systems show the feasibility of high-performance vswitches, the question remains whether we can achieve the same level of performance with a wider spectrum of vswitch use cases.

5.1.3 Network Functions

Network functions, including switches, routers and network middleboxes, provide network connectivity and value-added services to end hosts. For typical network functions implemented in software, most of its dataplane functionality – at least for packet-level services – tended to be embedded in the network stack of operating system kernel. However, in modern systems, key networking functions are scattered across the NIC, hypervisor, OS kernel, and user applications. This trend has been largely driven by the rise of cloud services. These services face stringent performance demands but kernel network stacks are notoriously slow and inefficient and this has led to many networking functions being "offloaded" to the NIC hardware [81]. Similarly, multi-tenancy requires safe communication between VMs which has led to hypervisors taking on a range of functions for traffic classification, shaping, access control, switching, and so forth. Moreover, new application needs have led to a growing set of networking protocols and features implemented in one or more locations (OS, NIC, hypervisor). Further complicating this picture is the recent emergence of specialized network facilities such as DPDK, netmap, and SRIOV: these systems address specific problems (e.g., packet I/O overhead, NIC slicing) or application contexts (e.g., NFV, network virtualization, and middlebox functionality) but their role with respect to legacy components remains unclear.

The problem is that the above evolution has taken place in a piecemeal manner, instead of being driven by an overarching coherent architecture for software network dataplane implementation. As we elaborate on in this chapter, this has resulted in: (i) mounting complexity, (ii) poor extensibility and, (iii) sub-optimal performance.

At a high level, the complexity comes from functions being scattered across disparate system components with no unified architecture to guide how network functions should be implemented, configured or controlled. The lack of a coherent architecture also leads to complex interactions between components that can be hard to diagnose or resolve.

The poor extensibility of existing implementations comes from the fact that functions implemented in NIC hardware are inherently less flexible and slower to evolve than software but, at the same time, software network functions that live in the kernel have also proven hard to evolve due to the complexity of kernel code.

Finally, the combination of complexity and poor extensibility has limited our ability to optimize the performance of legacy kernel network stacks and hypervisors. For example, numerous studies have used specialized network stacks to demonstrate the feasibility of achieving high performance traffic processing in software—e.g., [73] reports a 10-fold throughput improvement over typical kernel implementations—but these performance levels remain out of reach for commodity network stacks. A key issue here is that every network function dataplane is implemented from scratch without a common platform. This issue fundamentally constrains out ability to maximize reusability of dataplane "commonalities"; Best practices in design and implementation of software packet processing developed for one network function are not easily applicable to another.

5.2 BESS Design Overview

BESS is a programmable platform that allows external applications (controllers) to build a custom network dataplane, or pipeline. The pipeline is constructed in a declarative manner, being built as a dataflow graph of modules. Modules in the graph process perform module-specific operations on packets and pass them to next modules. The external controller programs the pipeline by deciding: what modules to use, how to configure individual modes, and how to interconnect them in the graph.

This section provides a high-level brief of BESS design. We set three main design goals (§5.2.1) and then provide details on the overall architecture (§5.2.2), the packet processing pipeline (§5.2.3), and the scheduler used for resource allocation (§5.4). In this section we only describe the design aspects of BESS, deferring implementation details on performance to §5.5.

5.2.1 Design Goals

1. Programmability and Extensibility: BESS must allow controllers to configure functionality to support a diverse set of uses. In particular, users should be able to compose and configure datapath functionality as required, and BESS should make it easy to add support for new protocols and services. Moreover, this additional effort should be reusable in a wide variety of usage scenarios. Clean separation between control and dataplane is also desired, so that an external controller can dynamically reconfigure the data path to implement user policies.

2. Resource scheduling as a first-class design component: Due to its inherent parallelism, hardware ASICs is in a better position to implement policies regarding resource allocation of link bandwidth, *e.g.,* "limit the bandwidth usage for this traffic class to 3 Gbps" or "enforce max-min fairness across all flows." BESS should provide flexible mechanisms to support a variety of policies on application-level performance. However, implementing these policies in software imposes a unique challenge for BESS—the processor[1] itself is a limited resource and must be properly scheduled to process traffic spanning multiple links, users, and applications.

3. Pay only for what you use: Finally, using BESS should not imply any unnecessary performance overheads. The cost of abstraction must be minimal, so that the performance would be comparable to the bare implementation of the same functionality without BESS would have exhibited.

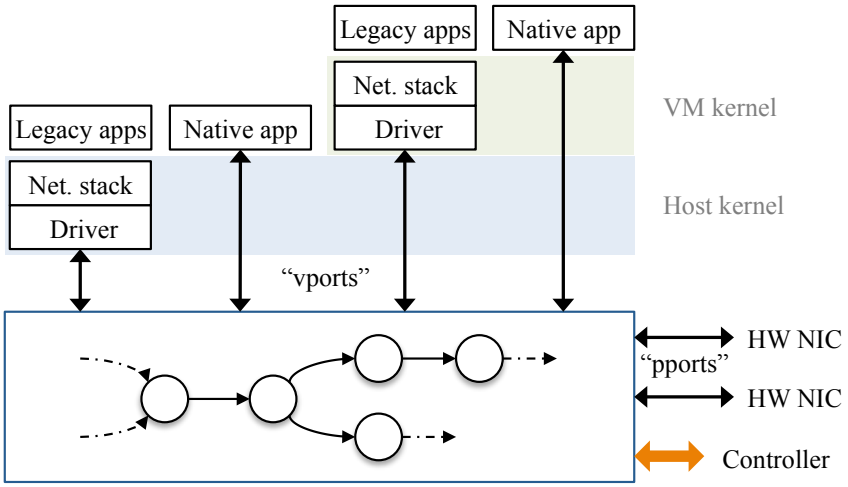


Figure 5.2: BESS architecture

5.2.2 Overall Architecture

Figure 5.2 shows the overall system architecture of BESS. The packet processing pipeline is represented as a dataflow (multi)graph that consists of modules, each of which implements a NIC feature. Ports act as sources and sinks for this pipeline. Packets received at a port flow through the pipeline to another port. Each module in the pipeline performs module-specific operations on packets. Our dataflow approach is heavily inspired by Click [65], although we both simplify and extend Click's design choices for BESS (§7.1).

BESS's dataflow graph supports two types of ports. (i) Virtual ports (*vports*) are the interface between BESS and upper-layer software. A vport connects BESS to a *peer*; a peer can either be the BESS device driver (which is used to support legacy applications relying on the kernel's TCP/IP stack) or a BESS-aware application that bypasses the kernel. A peer can reside either in the host[2], or in a VM. (ii) Physical ports (*pports*) are the interface between BESS and the NIC hardware (and are hence not exposed to peers). Each pport exposes a set of primitives that are natively implemented in its NIC hardware (*e.g.,* checksum offloading for specific protocols).

A vport pretends to be an ideal NIC port that supports all features required by its peer. The modules in the packet processing pipeline are responsible for actually implementing the features, either in software, by offloading to hardware, or with combination of both. BESS thus abstracts away the limitations of the underlying NIC hardware from peers, effectively providing a hardware abstraction layer (HAL) for NICs. BESS both allows for rapid prototyping of new NIC functionality in software, and is also useful in cases where hardware provides incomplete functionality (*e.g.,* by providing software implementation that allow a

[1]We only manage processor time used *within* the BESS dataplane; the processor time used by external applications themselves is out of BESS's control.

[2]We use the loosely defined term "host" to refer to a hypervisor, a Dom0, a root container, or a bare-metal system, to embrace various virtualization scenarios.

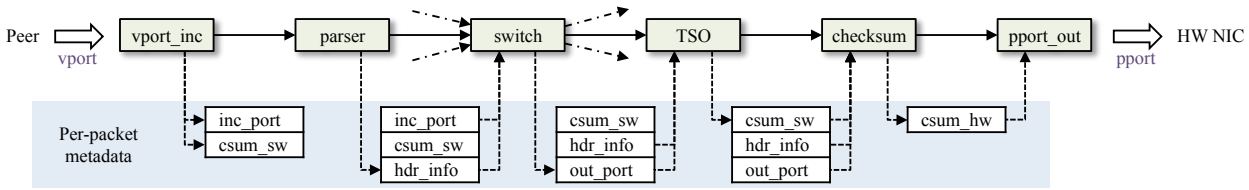


Figure 5.3: A pipeline example with parser, switch, TSO, and checksum offloading modules. Metadata evolves as a packet traverses the pipeline.

feature to be used with a new protocol) or insufficient capacity (*e.g.,* by using both software and hardware flow tables).

BESS provides a control channel that allows for a clean separation between the control and data plane. An explicit control channel allows an external controller to dictate data path policy, while BESS itself focuses on providing data-plane mechanisms for this policy. The control channel supports three types of operations: (1) updating the data path (e.g., adding/removing modules or ports), (2) configuring resource allocations (e.g., limiting CPU/bandwidth usages for applications), (3) managing individual modules (e.g., updating flow tables or collecting statistics). In this chapter we solely focus on the design and implementation of BESS, rather than the external controller.

5.2.3 Modular Packet Processing Pipeline

The BESS pipeline is composed of modules, each of which implements a NIC feature. An implementation of a module defines several handlers, including ones for processing packets and timer events. Each module instance may contain some internal state, and these instances are the nodes in BESS's data flow graph that specifies the order in which modules process packets (Figure 5.3). When a node has multiple outgoing edges (*e.g.,* classification modules), the module decides which of the edges a packet is sent out.

Often in-band communication between modules is desirable for performance and modularity. For example, a parser module performs header parsing and annotates the packet with the result as metadata, so that downstream modules can reuse this information. Along the pipeline, each packet carries its metadata fields abstracted as a list of key-value pairs. Modules specify which metadata fields they require as input and the fields they produce. Explicitly declaring metadata fields is useful in two ways. First, if a module in the pipeline requires a field that is not provided by any upstream modules, BESS can easily raise a configuration error. Second, any unused metadata field need not be preserved, and BESS can reduce the total amount of space required per-packet. Note that this optimization can be performed at configuration time and does not incur any runtime overhead.

Pipeline Example

We walk through a simple example to illustrate how packets are processed with metadata. In Figure 5.3, the tables on the edges show packet metadata at the point when a packet traverses the edge. The dotted arrows represent the input/output metadata fields for each module. The user configures the BESS pipeline so that transmitted packets are processed by (i) a switching service, (ii) TCP segmentation offload (TSO), and (iii) checksum offloading. The NIC in this example has no TSO functionality, and only supports checksum offloading for TCP/IPv4 packets. Nevertheless, the vport appears as a fully featured NIC to the peer, and provides both TSO and protocol-agnostic checksumming.

When the peer sends packets to the vport, each packet is annotated with its desired offloading behavior. For example, an annotation indicating that a packet requires checksum offloading is of the form "calculate checksum over byte range X using algorithm Y, and update the field at offset Z." The packet data and its annotations are packaged as a packet descriptor and pushed into the vport queue. In contrast to hardware NICs, the size and format of the descriptor are flexible and can change depending on the features exposed by the vport. Packet processing proceeds as follows.

1. **vport_inc** pulls a packet descriptor, creates a BESS packet buffer with the packet data, and adds metadata fields for the input port ID (`inc_port`) and checksum offloading description (`csum_sw`).

2. **parser** inspects the packet's L2–L4 header and records the results in `hdr_info`.

3. **switch** updates the MAC table using `inc_port` and `hdr_info`. Then it uses the destination address to determine the output edge along which the packet is sent. In this example the chosen edge goes to the TSO module.

4. **TSO** begins by checking whether the packet is a TCP packet larger than the MTU of `out_port`[3]. If so, the module segments the packet into multiple MTU-sized packets (and all metadata fields are copied), and updates `csum_sw` appropriately for each.

5. **checksum** uses `csum_sw` and `hdr_info` to determine if checksum calculation is required, and further if this needs to be done in software. When checksum computation can be carried out by the NIC containing `out_port`, the module simply sets `csum_hw` to "on", otherwise the module computes the checksum in software.

6. **pport_out** sends the packet to the NIC, with a flag indicating whether the hardware should compute the checksum, in the hardware-specific packet descriptor format.

[3]While in this dataflow graph the output port can be "inferred", explicit use of `out_port` allows greater flexibility: *e.g.,* allowing us to separate classification (which determines the `out_port`) from splitting (which diverts the packet flow in the pipeline).

This example highlights how BESS modules can flexibly implement NIC features and opportunistically offload computation to hardware. For ease of exposition we have only described the transmission path, the receive path is implemented similarly. Also, note a packet does not always need to flow between a vport and a pport: *e.g.,* virtual switching (between vports) [94] or multi-hop routing (between pports) [1].

5.3 Dynamic Packet Metadata

Packet metadata refers to any per-packet information conveyed with the packet data itself. Despite its integral role in software packet processing, usage of packet metadata in existing NFs is inefficient and inextensible. This section reviews the issues around packet metadata and describe how BESS solve the issues with dynamic attribute-offset assignment.

5.3.1 Problem: Metadata Bloat

Any packet processing software, including vswitches, defines a set of per-packet metadata attributes (or fields). Some of them convey essential information of a packet, thus used by all modules and the BESS runtime. Packet length, memory address, headroom and tailroom sizes, and reference counter are such examples. Some other attributes are rather optional and module-specific; they contain either contextual (e.g., input port ID, timestamp, etc.) or cached (e.g., checksum status, protocol and position of headers, etc.) information. Metadata attributes are typically represented as struct or class member fields, such as `struct sk_buff` in Linux and `struct mbuf` in BSD.

The set of metadata attributes tends to grow fast, as a network stack gets sophisticated and supports new features and protocols. For example, The network stack of Linux kernel version 4.2 stores 90 attributes per packet[4], which take 552 bytes spanning across 9 CPU cache lines (an order of magnitude larger than the payload of small-sized packets). This large per-packet footprint significantly hurts performance [41, 94]; it increases pressure on CPU cache and consumes more cycles for packet initialization with zeroes or attribute-specific initial values. This is wasteful given that only a few attributes would be actually used during the lifetime of each packet.

Click works around this metadata bloat issue by multiplexing multiple attributes ("annotations" in Click) in a per-packet scratchpad, 48-byte wide on its current implementation. The scratchpad space is statically partitioned for attributes. For example, an upstream element X stores 2-byte data at offset 16 for attribute 'foo', and a downstream element Y reads it from the offset for 'foo'. This approach can mitigate the growth of the metadata, yet introducing new issues:

1. Dependency: All modules using 'foo' must (implicitly) agree that it resides at fixed offset 16, at compile time.

[4] in `sk_buff` and `skb_shared_info`.

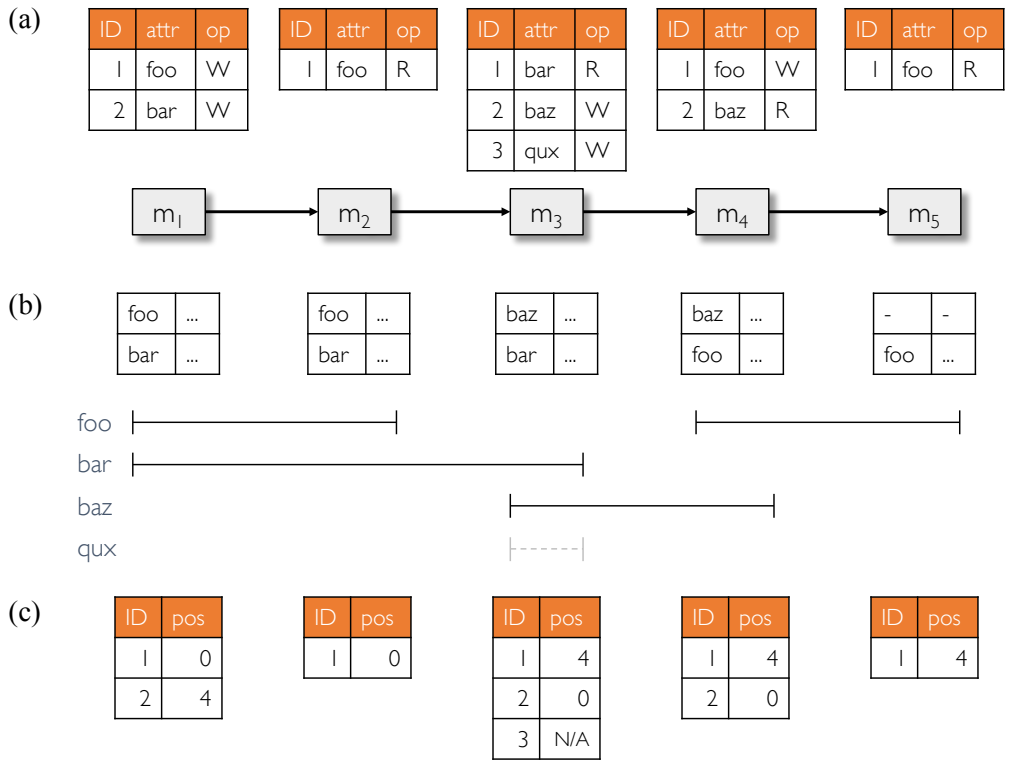


Figure 5.4: Per-packet metadata support in BESS

2. Aliasing: Some modules might happen to use the same data location to store a different attribute. They must not appear between X and Y otherwise may write over 'foo'.

3. Misconfiguration: Some control flow paths to Y may not initialize 'foo'. Y will read a garbage value for the attribute.

Because how modules utilize the scratchpad space is not visible to Click runtime, avoiding these issues is the sole responsibility of configuration writers (operators or external controllers, not module developers). Data-flow graph validation regarding metadata use is error-prone and must be done at the source-code level, thus at compile time. To avoid this issue one has to sacrifice modularity; i.e., writing one big, monolithic module rather than combining smaller, reusable modules with metadata.

5.3.2 Metadata Support in BESS

BESS solves the metadata bloat issue by making metadata an first-class abstraction. As shown in Figure 5.4(a), each module is required to specify the list of metadata attributes to be used and their size and access mode (e.g., *I read 'foo' and write 'bar' and 'baz'*). Attributes are identified by an opaque string; there is no need for global agreement on numeric identifiers of attributes. Since the metadata usage of modules is now visible, BESS runtime can easily

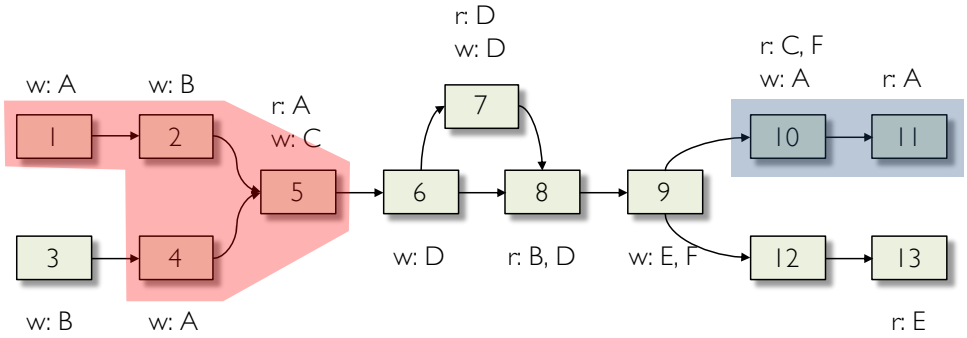


Figure 5.5: Two scope components of attribute `A`. "r" denotes that the corresponding module reads the specified attributes, and "w" is for writes.

check if there is any configuration error whenever the datapath is updated. For example, if not all paths to a module provide every required attribute, BESS warns that the module may read uninitialized values.

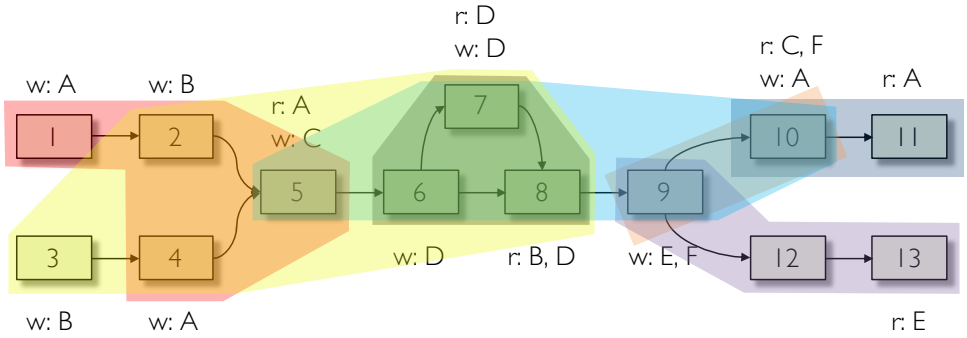
At the abstract level, every packet carries the value of metadata attributes in a table. The table contains a different set of attributes at each module, as depicted in Figure 5.4(b). Modules look up the per-packet table to access metadata attributes. Metadata access must be efficient, since it is a very frequent per-packet operation. Naive approaches, such as performing string matching on attribute names stored in the table, would be too costly in terms of both time and space.

For fast metadata access, BESS provides modules with an indirection array to locate metadata attributes (Figure 5.4(c)). The index is attribute ID, which is sequentially assigned within each module. The index of the array is an attribute ID, sequentially assigned within a module. The array stores the position of attributes, as a byte offset in the dynamic metadata section (currently 96 bytes) of each packet buffer. To access an attribute, a module first fetches the byte offset with the attribute ID and in turn accesses the data in the dynamic metadata section. Since the cost of the former part can be amortized over multiple packets, the performance cost of this indirect access is minimal.

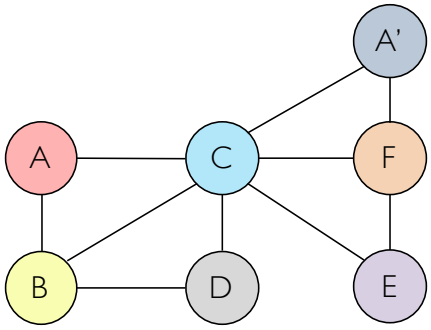
5.3.3 Attribute-Offset Assignment

Whenever the module graph topology changes, BESS (re)assigns attribute-offset mappings for each module. BESS runtime must ensure that attributes do not overlap in the metadata section for correctness. At the same time, BESS runtime is also responsible for efficient use of the limited per-packet space, to accommodate a large number of metadata attributes while minimizing CPU cache footprint.

In order to meet the requirements, BESS employs an assignment strategy based on two observations about Click's use of metadata attributes [65]. First, although Click has hundreds of modules, but only a few of them are actually used for a particular configuration. Therefore



(a) Scope components of all attributes.



(b) Scope graph constructed from the above. Each vertex corresponds to a scope component of an attribute, with the same color. Edges indicate overlaps in lifetime between two scope components.

Figure 5.6: Construction of the scope graph from calculated scope components. Best viewed in color.

we can multiplex attributes for the same offset, as long as it is safe to do so. Second, lifespan of each attribute can be quite short, allowing even more multiplexing. Attributes need space only for their lifetime, so after use we can reclaim the space to minimize the table size.

For simplicity, suppose that all attributes are 1 byte in size for now. The offset assignment starts with finding the lifetime of every metadata attribute. In the dataflow graph we define the lifetime of an attribute as its "scope component". Here, the scope component of an attribute is *the maximal connected component(s) in which every module must share the same offset for the attribute.* In the graph shown in Figure 5.5, for example, modules $\{1, 2, 4, 5\}$ forms a scope component of `A`. Module 3 is not part of the component since it is out of the lifetime of attribute `A`. Note that there is another scope component of attribute `A`, $\{10, 11\}$. This is a separate scope component of `A` because its value is newly (over)written by module 10, since the offset of `A` in the upstream modules do not matter. From the definition of scope component, we can easily deduce the following rules for given two different scope components, regardless of whether they are for the same attribute or not:

1. If they are overlapping, we *must* give them different offsets.

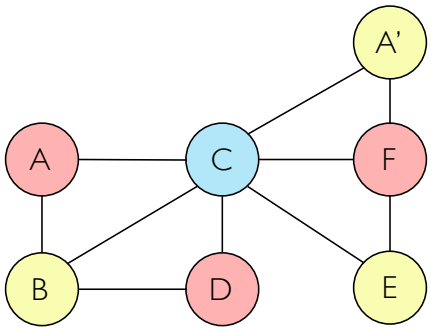


Figure 5.7: Optimal coloring for the scope graph shown in Figure 5.6(b) with three colors. In other words, for the given pipeline, three offsets are enough to accommodate all six attributes with multiplexing.

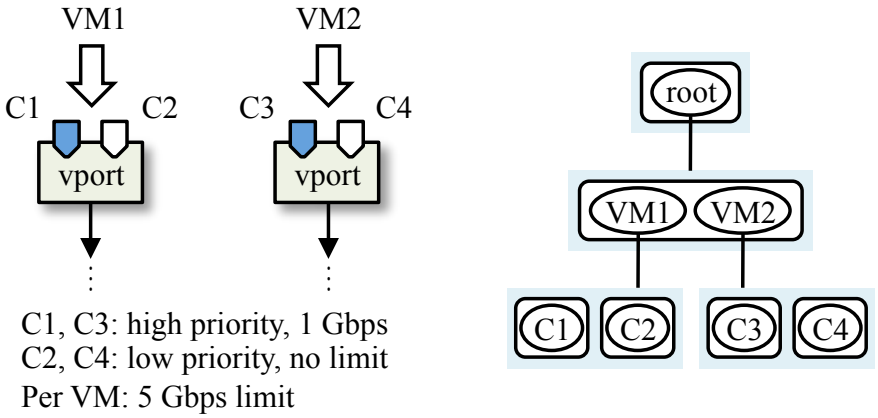
2. If they are disjoint, we *may* give them the same offset.

It is not difficult to see that these rules resemble the graph coloring problem, where adjacent vertices of a graph must be of different colors. To reduce attribute-offset assignment into graph coloring, we first construct a "scope graph" from the calculated scope components, as depicted in Figure 5.6. In the scope graph, each vertex corresponds to a scope component, and each edge represents if its two corresponding scope components overlap with each other. By definition, adjacent nodes mean that these two attributes are active at a given time, thus must have distinct offsets. From graph coloring on the scope graph, a color is 1-to-1 mapped to an offset, as we assumed all attributes are 1 byte. Figure 5.7 shows an optimal—*i.e.,* with the minimum number of colors—way to color the graph, where 3 colors/offsets are used for 6 attributes.

We note two practical issues arising here. First, the optimization version of graph coloring is an NP-hard problem; as the size of pipeline grows, calculating an optimal solution becomes intractable quickly. Second, unlike the assumption we made, each attribute may have an arbitrary size in general. It makes the problem a generalized version of graph coloring, as a vertex color is now defined as an offset range rather than a single number, and an edge represents overlapping offset ranges. In other words, this offset assignment problem is at least as difficult as graph coloring. Since finding an optimal solution is infeasible, BESS utilizes a heuristic with greedy coloring for offset assignment. In our experience, greedy coloring mostly gives us near-optimal solutions; we suspect that it is due to the nature of typical dataflow graph topology.

5.4 Resource Scheduling for Performance Guarantees

In contrast to NIC hardware, which is inherently parallel at the gate level, BESS needs a scheduler to decide what packet and module get to use the processor. For example, a simple



(a) Example policy (b) Class tree

Figure 5.8: An example of high-level performance policy with four traffic classes `C1`–`C4`, and its corresponding class tree representation. Each circle is either a class (leaf) or a class group (non-leaf). `VM1` and `VM2` have the same priority (within the same box).

form of scheduling[5] would involve using weighted round-robin scheduling to pick an input port, fetching a packet from this port and then processing it until it is sent out another port. This scheme is used by many software packet processing systems [41, 63, 65, 94, 104] for its simplicity. However, its crude form of fairness—the same number of packets across input ports—may not achieve the operator's desired *policy* for applications-level performance.

The ultimate goal of the BESS scheduler is to allow the operator to specify and enforce policies for applications. Instead of mandating a single policy (*e.g.,* priority scheduling) that must be used in all deployment scenarios, BESS provides a set of flexible mechanisms that can be easily composed to implement a broad set of high-level policies. The scheduler makes policy-compliant allocations of processor and bandwidth resources using the mechanisms.

Consider a typical policy example, as presented in Figure 5.8(a). In this example, each VM gets a fair share of bandwidth up to 5 Gbps. Each VM has two types of traffic: interactive traffic (`C1` and `C3`) and background traffic (`C2` and `C4`). The interactive traffic has higher priority but is limited to 1 Gbps per VM. Note that the policy includes: (i) fair sharing between VMs, (ii) rate limiting for each VM and its interactive traffic, (iii) fixed priorities between the traffic types, and (iv) hierarchical composition of the above. The design of BESS scheduler is centered around these four primitives to capture common policy patterns.

The scheduling unit of BESS data path execution is a traffic *class*, each of whose definition is flexible and not dictated by BESS. Possible class definitions include: input/output

[5] Software packet processing frameworks require two complementary types of scheduling: (i) packet scheduling, where a functional module (*e.g.,* `PrioSched` element in Click [65]) selects the next packet to process; and (ii) CPU scheduling, where the framework determines when and how often each module is executed (*i.e.,* gets processor time). For the former, we discuss how our approach differs from Click's in §7.1. In this section, we focus on the latter, as it has received less attention from the research community.

port, VM, tenant, protocol, L7 application, VLAN tag, IP prefix, and so on. The operator determines the scheduling discipline for traffic classes: *e.g.,* "`C1` has higher priority than `C2`" by setting appropriate scheduling parameters.

In BESS, every packet is mapped to one (and only one) of the traffic classes at any given time. The initial mapping of a packet to a class is "given" to BESS. A port consists of a set of input queues, each of which is mapped to a traffic class. Effectively, the peer (for a vport) or the hardware NIC (for a pport) declares the initial class of each packet by pushing the packet into its corresponding queue. The class of a packet may change on the fly; in the dataflow graph, the operator can specify *transformer edges*, which can associate a new class to packets that flow along the edge. Each transformer edge has a map $class_{old} \rightarrow class_{new}$. Class transformation is useful in cases where packet classes cannot be predetermined externally: e.g., (i) a hardware NIC has limited classification capability for incoming packets, or (ii) traffic classes are defined as output ports or input/output port pairs, so the class of a packet is logically unknown (thus a "unspecified" class) until its output port has been determined by a switch/classifier module.

Scheduling parameters are specified as a *class tree*, which is a hierarchical structure of traffic classes. For example, Figure 5.8(b) shows the class tree for the previous policy. The scheduler starts by examining the children of `root` and proceeds recursively until it finds an eligible leaf. First, assuming neither has exceeded its 5 Gbps limit, the scheduler chooses between `VM1` and `VM2` (they both have the same priority), using a weighted fair queuing mechanism. Without loss of generality, let us assume that `VM1` is chosen. Since `C1` has higher priority than `C2`, the scheduler will pick `C1`, unless it has exceeded its 1 Gbps limit or no packet is pending. Packet processing for `C1` begins by dequeuing a packet from one of its input queues. BESS measures processor time and bandwidth usage during packet processing. Usage accounting is again recursively done for `C1`, `VM1`, and `root`. We generalize this scheduling scheme and provide implementation details in §5.5.4.

The operator specify policies by providing a class tree, as shown in Figure 5.8(b). It is straightforward to translate the example policy into the corresponding class tree. For more complex policies, manual translation might be harder. In this case, an external controller would be required to compile a policy specified in a high-level language into BESS configuration (a dataflow graph and its class tree). Designing and implementing such a language and a compiler are left to future work. The controller is also responsible for performing admission control, which ensures that the number of applications using BESS does not exceed the available resources. Admission control can also be used in conjunction with priority scheduling to guarantee minimum bandwidth and processor allocations for applications.

5.5 Implementation Details

10/40 G Ethernet has become the norm for datacenter servers and NFV applications. In order to keep up with such high-speed links, BESS needs to process (tens of) millions of packets per second with minimal packet processing overheads. Latency is also critical;

recent advances in network hardware (e.g., [33]) and protocol design (e.g., [5]) have allowed microsecond-scale in-network latency, effectively shifting the bottleneck to end-host software [106]. BESS should therefore incur minimal processing delay and jitter.

Meeting both these performance requirements and our design goals is challenging. In this section, we describe the implementation details of BESS, with emphasis on its performance-related aspects.

5.5.1 Overview

As of today, BESS is implemented in 34k lines of C++ code, running with unmodified Linux and QEMU/KVM. We expect supporting other operating systems or virtualization platforms would be straightforward. To simplify development for us and module developers, BESS runs as a user-mode program on the host. This has performance implications; in our system, the minimum cost of a user-user (between an application process and BESS) context switch is 3 µs, while it is only 0.1 µs for a user-kernel mode switch. BESS uses one or more dedicated cores to eliminate the costs of context switching, as explained below. Interfacing with physical and virtual ports is done with DPDK [48]

BESS runs separate control threads alongside worker threads. The control threads communicates with external controllers via gRPC [38]. Any programming languages can be used to write a controller if only it has a gRPC client library. Some controller commands require synchronization across worker threads for consistent updates (*e.g.,* modifying the dataflow graph). If so, the control thread sets a barrier that blocks all worker threads until the command completes. BESS supports two types of commands: commands to the BESS framework itself (*e.g.,* create a new worker thread) and commands to individual modules (*e.g.,* add a static ARP entry). For the latter, BESS merely relays them as opaque messages to the destined module.

5.5.2 Core Dedication

BESS runs on a small number of dedicated cores—as will be shown later, one core is enough for typical workloads—for predictable, high performance. The alternative, running BESS threads on the same cores as applications (thus on every core) is not viable for us. Since BESS is a user-mode program, OS process scheduling can unexpectedly introduce up to several milliseconds of delay under high load. This level of jitter would not allow us to provide (sub)microsecond-scale latency overhead.

We dedicate cores by setting core affinity for BESS worker threads and prevent the host kernel from scheduling other system tasks on BESS cores with the `isolcpus` Linux kernel parameter. In addition to reducing jitter, core dedication has three additional benefits for BESS: (i) context switching costs are eliminated; (ii) processor cache is better utilized; (iii) inter-core synchronization among BESS threads is cheap as it involves only a few cores.

Core dedication allows us to make another optimization: we can utilize busy-wait polling instead of interrupts, to reduce latency further. A recent study reports 5–75 µs latency

overheads per interrupt, depending on the processor power management states at the moment [33]. In contrast, with busy-wait polling, the current BESS implementation takes less than 0.02 µs for a vport and 0.03 µs for a pport to react to a new packet. Polling leads to a small increase in power consumption when the system is idle: our system roughly consumes an additional 3–5 watts per idle-looping core.

We do not dedicate a core to the control thread, since control-plane operations are relatively latency-insensitive. The control thread performs blocking operations on sockets, thus consuming no CPU cycles when idle.

5.5.3 Pipeline Components

Physical Ports (pports)

We build on the Intel Data Plane Development Kit (DPDK) 17.11 [48] library for high-performance packet I/O. We chose DPDK over other alternatives [34, 41, 103] for two reasons: (i) it exposes hardware NIC features besides raw packet I/O; and (ii) it allows direct access to the NIC hardware without any kernel intervention on the critical path.

Each pport is associated with two module instances. **pport_out** sends packets to the hardware NIC after translating packet metadata into hardware-specific offloading primitives. BESS provides an interface for feature modules to discover the capabilities of each pport, so that modules can make decision about whether a feature can be partially or fully offloaded to hardware. **pport_in** receives incoming packets and translates their hardware-offload results into metadata that can be used by other modules.

Virtual Ports (vports)

A vport has a set of RX and TX queues. Each queue contains two one-way ring buffers; one for transmitting packet buffers and the other for receiving completion notification (so that the sender can reclaim the packet buffers). The ring buffers provide lock-free operations for multiple consumer/producer cores [79], to minimize inter-core communication overheads. A vport's ring buffers are allocated in a single contiguous memory region that is shared between BESS and the peer. Memory sharing is done using `mmap()` for host peers and IVSHMEM [71] for VM guest peers. When no interrupts are involved, communication via shared memory allows BESS to provide VM peers and host peers similar performance for packet exchange.

For conventional TCP/IP applications, we implement a device driver as a Linux kernel module that can be used by either hosts or guests. We expect that porting this device driver to other operating systems will be straightforward. The driver exposes a vport as a regular Ethernet adapter to the kernel. No modifications are required in the kernel network stack and applications. Our implementation is similar to virtio [107], but we support not only guest VMs but also the host network stack.

For kernel-bypass applications that implement their own specialized/streamlined network stack (*e.g.,* [14, 54, 73, 93]), we provide a user-level library that allows applications to directly

access vport queues, supporting zero copy if desired. In contrast, vport peering with the kernel device driver requires copying the packet data; providing zero-copy support for the kernel would require non-trivial kernel modifications, which are beyond the scope of this work.

When packets are transmitted from BESS to a peer, BESS notifies the peer via inter-core interrupts. This notification is not necessary when a peer sends packet because BESS performs polling. The peer can disable interrupts temporarily (to avoid receive livelock [83]) or permanently (for pure polling [24]).

NIC Feature Modules

To test and evaluate the effectiveness of our framework, we implemented a number of NIC features that are commonly used in datacenter servers: checksum offloading, TCP segmentation/reassembly and VXLAN tunneling (§6.1.1), rate limiter (§6.1.2), flow steering (§6.1.3), stateless/stateful load balancer (§6.1.4), time stamping, IP forwarding, link aggregation, and switching. Our switch module implements a simplified OpenFlow [76] switch on top of MAC-learning Ethernet bridging.

5.5.4 BESS Scheduler

The basic unit of scheduling in BESS is a traffic class (§5.4). Each class is associated with a set of queues (§5.5.6) and a per-core timer. Packets belonging to the class are enqueued on one of these queues before processing. The per-core timer is used to schedule deferred processing that can be used for a variety of purposes, *e.g.,* flushing reassembled TCP packets (LRO), interrupt coalescing, periodically querying hardware performance counter, scheduling packet transmission, etc. The scheduler is responsible for selecting the next class and initiating processing for this class.

BESS scheduling parameters are provided to the scheduler as a class tree (§5.4). A class tree is a hierarchical tree, whose leaves map to individual traffic classes while non-leaf nodes represent class groups. Each class group is a recursive combination of classes or other class groups. The scheduling discipline for each node in the class tree is specified in terms of a 5-tuple $\langle priority, limit_{bw}, limit_{cpu}, share, share\_type \rangle$, where each element specifies:

- *priority*: strict priority among its all siblings
- $limit_{bw}$: limit on the throughput (bits/s)
- $limit_{cpu}$: limit on processor time (cycles/s)
- *share*: share relative to its siblings with the same priority
- *share_type*: the type of proportional share: bits or cycles

The BESS scheduling loop proceeds in three steps:

Algorithm 1: Recursively traverse class tree to pick the next traffic class to service.

```
1 Node Pick(n)
2     if n is leaf then
3         return n ;                                  // We found a traffic class.
4     else
5         maxp ← max(n.children.priority);
6         group ← n.children.filter(
7                             priority = maxp);
8         next ←StrideScheduler (group);
9         return Pick (next);
```

1. **Pick next class.** The scheduler chooses the next class to service by recursively traversing the class tree starting at the root by calling the `pick` function (Algorithm 1) on the tree root. Given a node n, the `pick` function first checks (line 2) whether n is a leaf node (*i.e.,* a traffic class) in which case `pick` returns n. If n is not a leaf (and is thus a class group), *pick* find the highest priority level among n's children (*maxp*, line 5) and then finds the subset of its children assigned this priority (*group*, line 6). Finally, it uses stride scheduling [131] to select a child (*next*) from this subset (line 8) and returns the result of calling itself recursively on *next* (line 9).

2. **Servicing the class.** Once `pick` has returned a class c to the scheduler, the scheduler checks if c has any pending timer events. If so, the scheduler runs the deferred processing associated with the timer. If no timer events are pending, the scheduler dequeues a batch of packets (§5.5.5) from one of the classes queues (we use round-robin scheduling between queues belonging to the same class) and calls the receive handler for the first module—a port module or a module next to a transformer edge—in the packet processing pipeline (§5.2.3) for c. Packet processing continues until all packets in the batch have been either enqueued or dropped. Note that within a class, we handle all pending timer events before processing any packets, *i.e.,* timer events are processed at a higher priority, so that we can get high-accuracy for packet transmission scheduling, as shown in §6.1.2.

3. **Account for resource usage.** Once control returns to the scheduler, it updates the class tree to account for resources (both processor and bandwidth) used in processing class c. This accounting is done with the class c and all of its parents up to the root of the class tree. During this update, the scheduler also temporarily prunes the tree of any nodes that have exceeded their resource limits. This pruning (and grafting) is done with the token bucket algorithm.

As stated in §5.4, sometimes assigning a packet to the correct class might require BESS to execute one or more classification modules before sending a packet out a transformer edge.

For accurate resource accounting in this case, BESS associates an implicit queue with each transformer edge so that packets whose class has been changed are enqueued and wait for the new class to be scheduled before further processing.

5.5.5 Packet Buffers and Batched Packet Processing

Packet Buffer

BESS extends the DPDK's packet buffer structure (`rte_mbuf`) [48] by reserving an area in each buffer to store metadata. We rely heavily on scattered packet buffers (*i.e.,* non-contiguous packet data) to avoid packet copy, for operations such as segmentation/reassembly offloading and switch broadcasting.

Per-Packet Metadata

A naive implementation of metadata (§5.2.3), *e.g.,* using a hashmap of *string→value*, can have significant performance penalty, as in BESS tens of modules may process tens of millions of packets per second. To avoid the performance penalty, previous software packet processing systems use either a static set of metadata fields as `struct` fields (*e.g.,* BSD mbuf [80]) or per-packet scratchpad where all modules have to agree on how each byte should be partitioned and reused (*e.g.,* Click [65]). Both approaches are not only inefficient in space (thus increasing CPU cache pressure) and inextensible, but also are error-prone.

BESS takes advantage of the explicit declaration of metadata fields by each module (§5.3). Since all fields are known ahead of time, offsets for metadata fields in a pipeline can be precomputed, and BESS provides the offsets to every module during the pipeline (re)configuration phase. Subsequently, modules can access a metadata field by reading from its corresponding offset from the packet buffer. In the current implementation BESS reserves a 96-byte scratchpad for each packet to accommodate metadata fields.

Pervasive Batching

Packets in BESS are processed in batches, *i.e.,* packet I/O from/to ports is done in batches, and packet processing modules operate on batches of packets, rather than individual packets. Batching is a well-known technique for improving code/data locality and amortizing overheads in software packet processing [14, 24, 41, 62, 103]. Specifically, batching amortizes the cost of (i) remote cache access for vports, (ii) hardware access over the PCIe bus for pports, and (iii) virtual function calls for module execution. BESS uses a dynamic batch size that is adaptively varied to minimize the impact on latency as in IX [14]: the batch size grows (to a cap) only when BESS is overloaded and there is queue buildup for incoming packets.

When combined with the modular pipeline of BESS, batch processing provides additional performance benefits. Since the processing of a packet batch is naturally "staged" across modules, cache miss or register dependency stalls during processing one packet can be likely

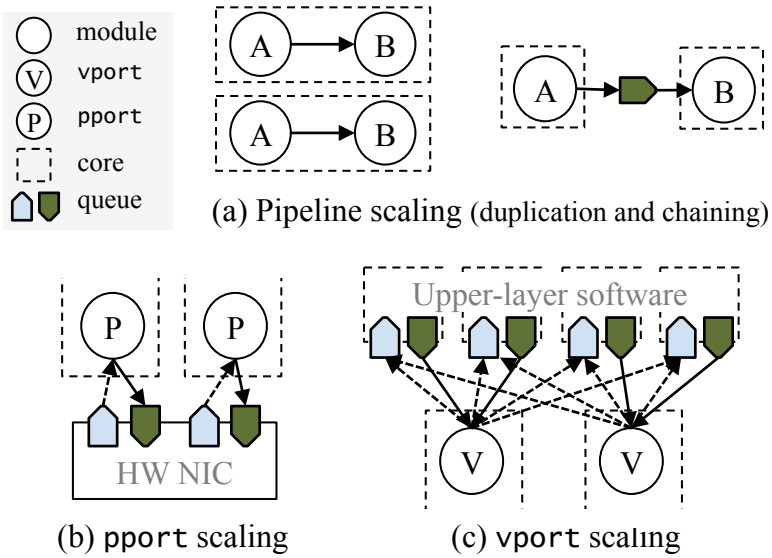


Figure 5.9: BESS multi-core scaling strategies. The example assumes two BESS cores and four cores for the upper-layer software.

hidden by processing another packet in the batch on an out-of-order execution processor [58]. It also reduces the cost of virtual function call.

A packet batch is simply represented as an array of packet buffer pointers. When packets in a batch need to take different paths in the pipeline (*e.g.,* classifier or switch), the module can simply split the batch by moving pointers from one array to another without incurring significant overheads.

In case there are many branches in the pipeline, there might be a number of batches due to branching, each holding only a few packets. In terms of performance small batches are suboptimal, since amortization of per-packet cost becomes much less effective. To mitigate this issue, BESS scheduler tries to combine multiple small batches into one large batch at every input gate of modules. For maximum effectiveness, modules are scheduled in the topological order of the dataflow graph, *i.e.,* , a module is scheduled only after all of its upstream modules have been scheduled, in order to combine as many batches as possible.

5.5.6 Multi-Core Scaling

While BESS running on a single core often provides enough horse power to drive the common network workload of a datacenter server (a 40 G link or higher with a typical packet size distribution, see §5.6.3), BESS can scale on multiple cores to sustain more challenging workloads. Here we consider three aspects of multi-core scalablity: individual modules, whole pipeline, NIC ports, and scheduler.

Modules

BESS module classes can explicitly declare if its implementation is thread-safe. If not, BESS conservatively assumes that it is unsafe to run on multiple threads. Whenever the pipeline is updated, BESS runs a constraint checker, which checks if thread-unsafe modules may possibly be accessed by multiple worker threads. When it happens—it typically implies a logic error in the external controller—BESS rejects the pipeline change. Alternatively, it would be possible to automatically apply a spin-lock for such thread-unsafe modules. We are adding this feature to the current implementation.

BESS does not dictate what strategy a module class should take for thread safety. Many modules have taken different approaches to thread safety and/or linear performance scalability, including locks, atomic variables, lock-less algorithms, and Transactional Memory [126].

Pipeline

In order to run a logical pipeline on multiple cores, we need to decide which part of the dataflow graph should be run by which worker thread(s). Figure 5.9(a) illustrates how it can be done. BESS provides two means to scale the packet processing pipeline. One scheme is *duplication*, each core runs the identical pipeline of modules in parallel. One can either replicate the modules or have multiple worker threads run on the same module instances. This approach is very well suitable for stateless modules in particular, since embarrassingly parallel operations scale well with multiple threads.

The other scheme is *chaining*, where the pipeline is partitioned with a queue module. The queue is used for one worker thread to hand over packets to another worker. This scheme is often used when thread-unsafe modules are not easily parallelizable. In terms of performance, this scheme yields lower throughput because of two reasons: i) packets cross core boundaries, increasing cache coherence traffic, and ii) load may be imbalanced (*e.g.,* in Figure 5.9(a), module A takes more cycles to process a packet than B, thus leaving B underutilized).

One scheme is not always better than the other, as their applicability and resulting performance highly depend on various factors: cache usage of modules, synchronization overhead, number of BESS cores, etc. BESS does not try to optimize pipeline scaling on behalf of the external controller. Rather, BESS simply provides mechanisms to configure the pipeline for duplication and chaining so that external controller can choose a policy to balance between performance, service-level objectives, and complexity. One can also try a hybrid of the both schemes. For example, in our NFV research system [88], we chained "I/O" and "processing" in the pipeline, where each of them consists a group of worker threads with duplication.

NIC ports

When a pport is duplicated across cores, BESS leverages the multi-queue functionality of the hardware NIC as shown in Figure 5.9(b). Each BESS core runs its own RX/TX queue

pair, without incurring any cache coherence traffic among BESS cores. Similarly, BESS creates multiple queue pairs for a vport so that the peer, the other endpoint of the virtual link, itself can linearly scale. By partitioning the incoming (from the viewpoint of BESS) queues of vports and pports, BESS preserves in-order packet processing on a flow basis, provided that peers and hardware NICs do not interleave a flow across multiple queues.

Scheduler

Each BESS worker thread, pinned to its dedicated CPU core, runs an independent scheduling loop with its own class tree (§5.4). This is our design choice in favor of performance; sharing a single, global class tree among worker threads would have caused too much inter-core synchronization cost, nullifying the point of multiple cores. By having a separate scheduler and a class tree, a worker thread can make scheduling decisions on its own. This is crucial to achieve linear performance scalability with the number of CPU cores.

On the other hand, running a separate scheduler on each work thread means that only one scheduler may schedule a traffic class and measure its resource use at any given time. In case we need to enforce resource allocation limit for certain type of traffic across multiple cores, an external controller would be required in the feedback loop. The controller needs to monitor the aggregate resource usage of relevant traffic classes and "rebalance" the limit of each.

5.6 Performance Evaluation

The main promise of BESS is to provide a flexible framework for implementing various NIC features, without sacrificing performance. In this section, we focus on the overheads of BESS framework itself, without considering the performance of individual feature modules. We quantify the overheads by measuring how fast BESS performs as a NIC, in terms of end-to-end latency (§5.6.2), throughput and multi-core scalability (§5.6.3).

5.6.1 Experimental Setup

We use two directly connected servers, each equipped with two Intel Xeon 2.6 GHz E5-2650v2 processors (16 cores in total), 128 GB of memory, and four Intel 82599 10 GbE ports with an aggregate bandwidth of 40 Gbps. We disable the CPU's power management features (C/P-states) for reproducible latency measurement [33]. We use unmodified Linux 3.14.16 (for both host and guest), QEMU/KVM 1.7.0 for virtualization, and ixgbe 3.19.1-k NIC device driver in the test cases where BESS is not used.

5.6.2 End-to-End Latency

Figure 5.10 shows the end-to-end, round-trip latency measured with UDP packets (TCP results are similar). In T1, the application performs direct packet I/O using a dedicated

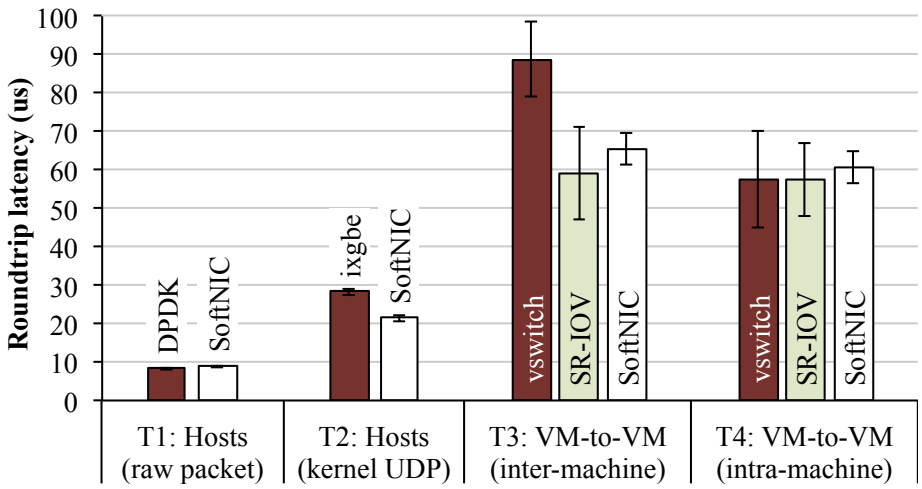


Figure 5.10: Round-trip latency between two application processes.

hardware NIC with DPDK (8.22 µs) or a vport (8.82 µs), thus bypassing the kernel protocol stack. BESS adds a small latency overhead of 0.6 µs per round trip, or 0.15 µs per direction per server (one round trip involves BESS four times). We posit that the advantage of using BESS—allowing multiple kernel-bypass and conventional TCP/IP applications to coexist without requiring exclusive hardware access—outweighs the costs.

T2–4 shows the latency for conventional applications when using the kernel TCP/IP support, measured with the netperf UDP_RR [84] test. Surprisingly, for the non-virtualization case (T2), the baseline (ixgbe) latency of 28.3 µs was higher than BESS's 21.4 µs. This is due to a limitation of 82599; when LRO is enabled, the NIC buffers incoming packets (thus inflating latency), even if they do not require reassembly. When LRO is disabled, latency decreases to 21.1 µs, which comes very close to that of BESS.

With server virtualization, T3[6], we compare BESS with virtual switching in the host network stack (vswitch) and hardware NIC virtualization (SR-IOV). As expected, using a vswitch incurs significant latency overhead because packets go through the slow host network stack four times (vs. two times in T4) in a single round trip. For SR-IOV and BESS, packets bypass the host network stack. When compared with the bare-metal case (T2), both exhibit higher latency, because packet I/O interrupts to the VMs have to go through the host kernel. We expect that the latency for VMs using SR-IOV and BESS will be close to bare-metal with recent progresses in direct VM interrupt injection [36, 44], and the latency of BESS will remain comparable to hardware NIC virtualization.

In summary, the results confirm that BESS does not add significant overhead to end-to-end latency for both bare-metal and virtual machines. We conclude that BESS is viable even for latency-sensitive applications.

[6]For completeness, we also show T4 where two VMs reside on the same physical machine.

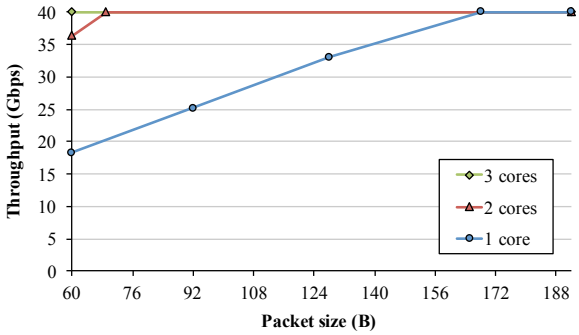


Figure 5.11: BESS throughput and multi-core scalability.

5.6.3 Throughput

We demonstrate that BESS sustains enough throughput on a single core to support high-speed links and scales well on multiple cores. For the experiment, we could not use conventional TCP/IP applications due to the low performance (roughly 700 kpps per core) of Linux TCP/IP stack, which is not enough to saturate BESS. Instead, we write a simple application that performs minimal forwarding (packets received from a port are forwarded to another port) with the BESS vport raw packet interface. The application runs on a single core and is never the bottleneck since effectively the heavy-weight packet I/O is offloaded to BESS.

Figure 5.11 depicts the bidirectional throughput with varying packet sizes as we increase the number of BESS cores. In the worst case with 60 B minimum-sized Ethernet packets, BESS on a single core can sustain about 27.2 Mpps (18.3 Gbps) for both RX and TX directions simultaneously, fully saturating our 40 G link capacity with 168 B packets or larger. With multiple BESS cores, the throughput almost scales linearly. Considering the average packet size of 850 B in datacenters [16], we conclude that *BESS on a single core provides enough performance to drive a 40 G link* in realistic scenarios. For higher speed links (e.g., 100 Gbps for end hosts in the future) or applications with emphasis on small-packet performance (e.g., VoIP gateway), BESS needs to run on multiple cores.

Given that the number of cores in a typical datacenter server is large (16-32 as of today) and keeps pace with the link speed increase, we expect the required number of BESS cores for future high-speed links will remain relatively small. We also note that this small investment can bring huge net gain—-as we will see in the following macrobenchmarks—because BESS effectively takes over the burden of implementing NIC features from the host network stack, running them with much higher efficiency.

5.6.4 Application-Level Performance

We examine the performance impact of BESS on end-to-end application performance using memcached [77]. We use memaslap [3] as a load generator, with 64-byte keys, 1,024-byte values, and 9:1 get/put request ratio. We use 10k client TCP connections, which give similar

		Throughput (ops/s)	CPU usage
Bare-metal	ixgbe	593,345	945%
	BESS	608,974	949%
Virtualized	vswitch	185,060	1,009%
	SR-IOV	138,297	432%
	BESS	344,980	1,034%

Table 5.1: Memcached throughput and its corresponding system-wide CPU usage (all 16 cores are used thus 1,600% is the maximum). The BESS cases includes its own overhead of 100%, as BESS dedicates a single core. BESS heavily boosts the performance of network I/O under virtualization.

load to the network stack for both the baseline and BESS cases (§6.1.3). Table 5.1 shows the system-wide CPU usage (not fully utilized because of the global lock in memcached) and the throughput.

With virtualization, the slowpath processing of `vswitch` is 57% slower in terms of efficiency (i.e., CPU cycles per operation), as traffic goes through the host network stack incurring high CPU overheads. In contrast, BESS and SR-IOV equally performs well in terms of CPU cycles per operation by virtue of host bypass. However, the absolute throughput of BESS is much higher than that of SR-IOV, because of the limitation of the SR-IOV of 82599 NIC and its device driver; it only supports a single RX/TX queue pair for each VM, severely limiting the multi-core scalability of the VM's network stack. We suspect that this limitation is due to the lack of native per-VM rate limiters (cf., §6.1.2). 82599 only has single-level per-queue rate limiters, mandating use of a single queue to emulate per-VM rate enforcement.

We note that this kind of artifacts of SR-IOV is fundamental to its design and implementation. As SR-IOV is implemented by statically partitioning the hardware NIC resources (e.g., NIC queues), implementation of feature logics need to be physically duplicated for each VM. Otherwise, features would be unusable by VMs or lack per-VM control. The complete software bypass of SR-IOV imposes another flexibility issue; the hardware defines the boundary what the system can do as there is no way to introduce software augmentation. The host-based software approach does not suffer from such flexibility issues – since its behavior is not hard limited the capability/capacity of underlying hardware NICs – although its low performance is problematic. We address this performance issue with BESS.

Chapter 6

Applications of BESS

6.1 Case Study: Advanced NIC features

BESS provides an effective platform to implement a wide variety of NIC features that are either not readily available or cannot be fully implemented in hardware, while providing higher performance than achievable by a software implementation in host network stacks. This section highlights the programmability and performance of BESS with various NIC feature examples.

6.1.1 Segmentation Offloading for Tunneled Packets

Many NIC features perform protocol-dependent operations. However, the slow update cycle of NIC hardware cannot keep up with the emergence of new protocols and their extensions. This unfortunate gap further burdens already slow host network stacks [96], or restricts the protocol design space (e.g., MPTCP [46, 98] and STT [66]). BESS is a viable alternative to the problem as it can be easily reprogrammed to support new protocols and their extensions while minimizing performance overheads.

As an example, we discuss TCP segmentation offloading (TSO for sender-side segmentation and LRO for receiver-side reassembly) over tunneling protocols, such as VXLAN [72], NVGRE [122], and Geneve [37]. These tunneling protocols are used to provide virtual networks for tenants over the shared physical network [66]. Most 10 G NICs do not support segmentation offloading for inner TCP frames, since they do not understand the encapsulation format. While 40 G NICs have begun supporting segmentation for tunneled packets, VM traffic still has to go through the slow host network stack since NICs lack support for encapsulating packets itself and IP forwarding (for MAC-over-IP tunneling protocols, *e.g.,* VXLAN, and Geneve).

We compare TCP performance over VXLAN on two platforms: the host network stack (Linux) and BESS. Adding VXLAN support to the regular (non-tunneled) TCP TSO/LRO modules in BESS was trivial, requiring only 70 lines of code modification. The results shown in Table 6.1 clearly demonstrate that segmentation *onloading* on BESS achieves much

	Throughput	BESS	QEMU	Host	Guest	Total
Linux	**14.4** Gbps	-	-	475.2	242.6	**717.8**
BESS	**40.0** Gbps	200.0	66.6	9.8	186.1	**462.4**

(a) Sender-side CPU usage (%)

	Throughput	BESS	QEMU	Host	Guest	Total
Linux	**14.4** Gbps	-	-	771.7	387.2	**1158.9**
BESS	**40.0** Gbps	200.0	80.5	21.9	179.0	**481.4**

(b) Receiver-side CPU usage (%)

Table 6.1: TCP throughput and CPU usage breakdown over the VXLAN tunneling protocol. 32 TCP connections are generated with the netperf TCP_STREAM test, between two VMs on separate physical servers. The current implementation of BESS injects packet I/O interrupts through QEMU; we expect that the QEMU overheads be eliminated by injecting interrupts directly into KVM. Overall, BESS outperforms the host network stack, by a factor of 4.3 for TX and 6.7 for RX in terms of throughput per CPU cycle.

higher throughput at lower CPU overhead. BESS running on two cores[1] was able to saturate 40 Gbps, while Linux's throughput maxed out at 14.4 Gbps even when consuming more CPU cycles. In terms of CPU efficiency, *i.e.,* bits per cyle, BESS is 4.3× and 6.7× more efficient than Linux for TX and RX respectively.[2]

We note that there is a widely held view that hardware segmentation offloading is indispensable for supporting high-speed links [66, 96, 97, 98]. Interestingly, our results show that software approaches to TSO/LRO can also support high-speed links, as long as the software is carefully designed and implemented.

6.1.2 Scalable Rate Limiter

Many recent research proposals [9, 56, 95, 105, 114] rely on endhost-based rate enforcement for fair and guaranteed bandwidth allocation in a multi-tenant datacenter. These systems require a large number (< 1,000s) of rate limiters, far beyond the capacity of commodity NICs (*e.g.,* Intel's 82599 10 G NIC supports up to 128 rate limiters, and XL710 40 G

[1] Unlike the regular TCP TSO/LRO with which BESS can saturate 40 Gbps on a single core, we needed two BESS cores for VXLAN. This is because BESS has to calculate checksum for the inner TCP frame in software; Intel 82599 NICs do not support checksum offloading over an arbitrary payload range.

[2] The asymmetry between TX and RX is due to the fact that the host network stack implements TCP over VXLAN segmentation for the sender side [136], but reassembly at the receiver side is currently not supported, thus overloading the host and VM network stacks with small packets.

Per-flow rate (Mbps)	Number of flows	SENIC (µs)	BESS (µs)
1	1	-	0.8
1	500	7.1	1.4
1	4096	N/A	2.0
1	8192	-	6.5
10	1	0.23	1.4
10	10	0.24	1.4
10	100	1.3	1.6
10	1000	-	4.4
100	1	0.087	1.1
100	10	0.173	1.4
100	100	-	1.4
1000	1	0.161	1.1
1000	3	0.191	1.1
1000	10	-	0.6

Table 6.2: Accuracy comparison between BESS and SENIC rate limiters, with the standard deviation of IPGs. The SENIC numbers are excerpted from their NetFPGA implementation [97, Table 3].

NIC supports 384). Software rate limiters in host network stacks (*e.g.,* Linux tc [47]) can scale, but their high CPU overheads hinder support for high-speed links and precise rate enforcement [97]. We show that rate enforcement with BESS achieves both scalability and accuracy for high-speed links.

We conduct an experiment with 1,000 concurrent UDP flows, whose target rate ranges between 1–11 Mbps with 10 kbps steps. The measured accuracy for each flow—the ratio of the measured rate to the target—is higher than 0.9999.

Another important metric is microscopic accuracy, *i.e.,* how evenly flows are paced at the packet level, since precise packet burstiness control on short timescales is essential for achieving low latency in a datacenter [53, 90]. Table 6.2 shows the standard deviation of inter-packet gaps (IPG) measured at the receiver side. Since we collect IPG measurements on a server, instead of using specialized measurement hardware, our IPG numbers are overestimated due to the receiver's measurement error. Nevertheless, BESS achieves about 1-2 µs standard deviation across a wide range of scenarios.

As a point of comparison[3], we also show the results from SENIC [97] as a state-of-the-art hardware NIC. The hardware-based real-time behavior allows SENIC to achieve a very low standard deviation of 100 ns with 10 or less flows, but its IPG variation increases as the number of flows grows. At the largest data point presented in [97, Table 3], 500 flows

[3]We omit the Linux tc and Intel 82599 NIC results, as they fail to sustain the aggregate throughput or do not support enough rate limiters, respectively.

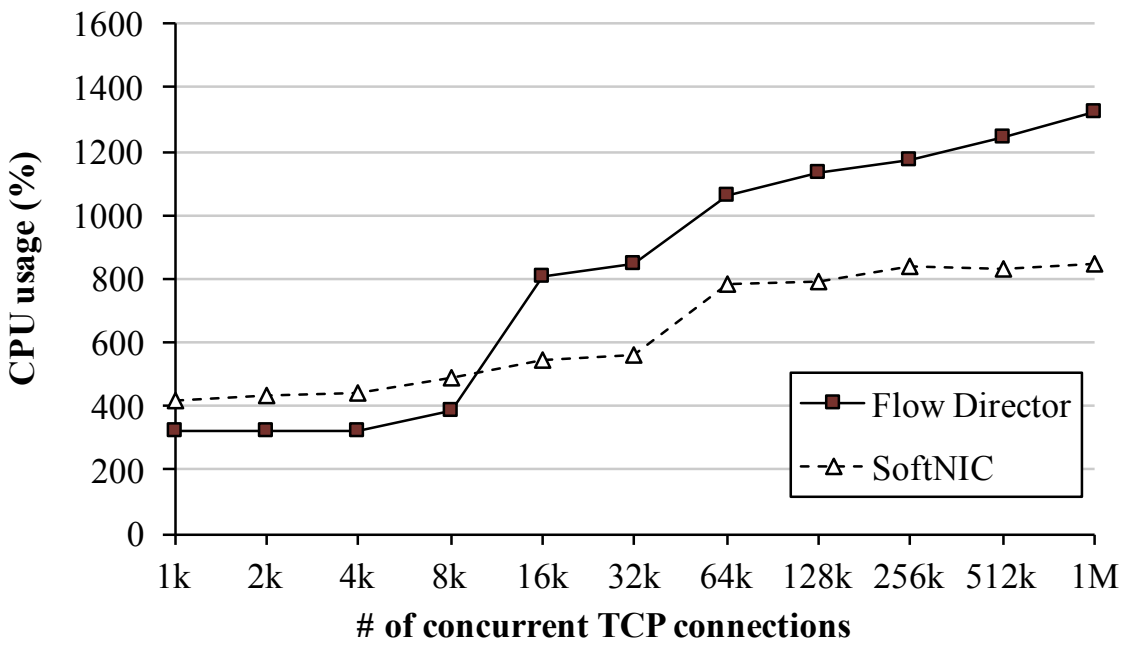


Figure 6.1: System-wide CPU usage to sustain 1M dummy transactions per second. The bump between 32k and 64k is due to additional TCP packets caused by delayed ACK. With BESS, the system scales better with high concurrency.

at 1 Mbps, the standard deviation of SENIC is 7.1 µs, which is higher than BESS's 1.4 µs. This is because the packet scheduler in SENIC performs a linear scan through the token bucket table on SRAM, requiring 5 clock cycles per active flow. While we are unaware of whether this linear scanning is due to hardware design limitations or implementation difficulties, BESS can achieve near-constant time with a time-sorted data structure whose software implementation is trivial. We conclude that BESS scales well to thousands of rate limiters, yet is fast and accurate enough for most practical purposes.

6.1.3 Packet Steering for Flow Affinity

Ensuring flow affinity—collocating TCP processing and application processing on the same core is known to be crucial for the performance TCP-based server applications. Existing software solutions to flow affinity [42, 92] restrict the application programming model, incurring multiple limitations: each application thread needs to be pinned to a core, connections should not be handed over among cores, and applications may require non-trivial code modifications.

As a more general solution without these limitations, Intel 82559 NICs support a feature called Flow Director, which maintains a flow-core mapping table so that the NIC can deliver incoming packets to the "right core" for each flow [51]. However, the table size is limited to 8k flow entries to fit in the scarce on-chip memory. Applications that require higher concurrency

(*e.g.,* front-end web servers and middleboxes often handle millions of concurrent flows) cannot benefit from this limited flow table. In BESS, we implement a module BESS that provides the same functionality but supports virtually unlimited flow entries by leveraging system memory.

Figure 6.1 shows the effectiveness of Flow Director and its BESS counterpart. We use a simple benchmark program that exchanges 512 B dummy requests and responses with a fixed rate of 1M transactions per second. We vary the number of concurrent TCP connections and measure the total CPU usage of the system. When there are a small number of concurrent connections, BESS's overhead is slightly higher due to the cost of the dedicated BESS core[4]. Once the number of connections exceeds 8k and the hardware flow table begins to overflow, Flow Director exhibits higher CPU overheads due to increased cache bouncing and lock contention among cores. In contrast, BESS shows a much more gradual increase (due to CPU cache capacity misses) in CPU usage with high concurrency. With 1M connections, for instance, BESS effectively saves 4.8 CPU cores, which is significant given that BESS is a drop-in solution.

6.1.4 Scaling Legacy Applications

Many legacy applications are still single-threaded, as parallelization may require non-trivial redesign of software. BESS can be utilized to scale single-threaded network applications, given that they do not need state to be shared across cores. We use Snort [117] 2.9.6.1, an intrusion prevention system as an example legacy application to show this scaling. There are two requirements for scaling Snort: (i) the NIC must distribute incoming packets across multiple Snort instances; (ii) moreover, such demultiplexing has to be done on a flow basis, so that each instance correctly performs deep-packet inspection on every flow. While we can meet these requirements with receive-side scaling (RSS), an existing NIC feature in commodity hardware NICs, it is often infeasible; RSS requires that any physical ports used by Snort not be shared with other applications.

The flexible BESS pipeline provides a straightforward mechanism for scaling Snort. BESS provides a backwards-compatible mechanism to distribute traffic from a physical link between multiple instances. To do this, we create a vport for each Snort instance and connect all vports and a pport with a load balancer module. The load balancer distributes flows across vports using the hash of the flow's 5-tuple. With this setup, each Snort instance can transparently receive and send packets through its vport. Figure 6.2 shows the 99th percentile end-to-end (from a packet generator to a sink) latency, using packet traces captured at a campus network gateway. We find that even at 500 Mbps a single instance of Snort (SINGLE) has a tail latency of approximately 366 ms, while using four Snort instances with our hash-based loadbalancer (HASH) limits latency to 57 µs (a $6,000\times$ improvement).

Furthermore, BESS allows us for rapid prototyping of more advanced load balancing schemes. We implemented an adaptive load balancer, ADAPTIVE, which tracks load at

[4]For a fair comparison, we always account 100% for the BESS case to reflect its busy-wait polling. Throughout the experiment, however, the actual utilization of the BESS core was well below 100%.

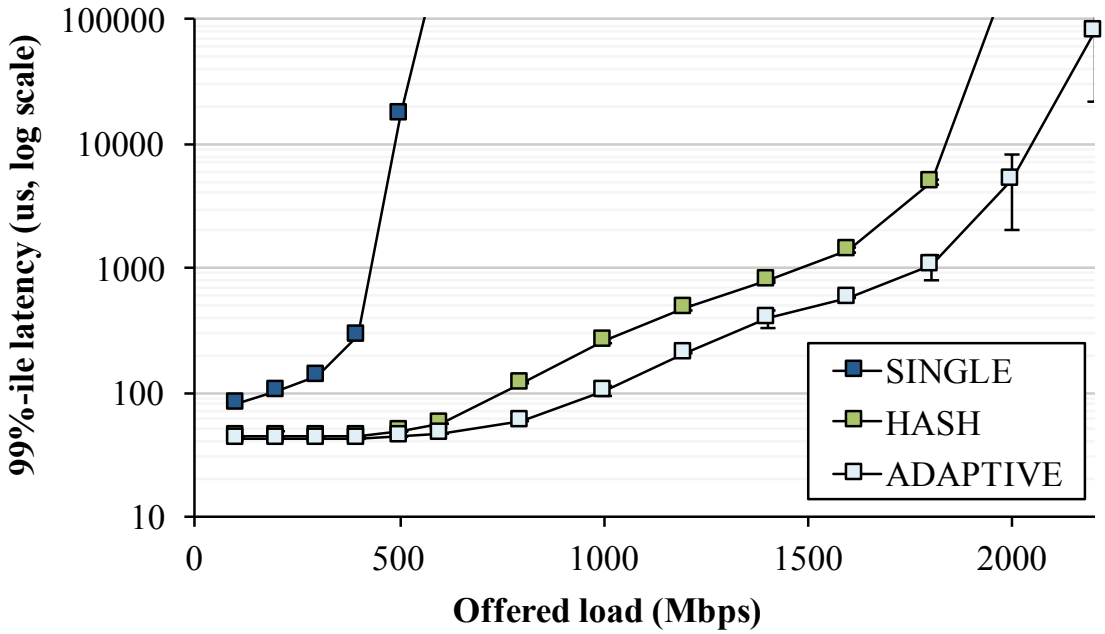


Figure 6.2: Snort 99%-ile tail latency with a single instance without BESS (SINGLE), four instances with static (HASH) and dynamic (ADAPTIVE) load balancing with BESS. Note that the absolute low throughput is due to the application bottleneck, not BESS.

each instance and assigns new flows to the least loaded vport. As compared to HASH, this scheme mitigates transient imbalance among instances, prevents hash-collision based attacks [125], and retains flow stickiness upon dynamic change in number of instances. Our current heuristic estimates the load of each port using "load points"; we assign a vport 10,000 load points for its new flow, 1,000 points per packet, and 1 point per byte. When a new flow arrives, we assign it to the vport which accumulated the fewest load points in the last 1 ms time window. The adaptive load balancer maintains a flow table for flow stickiness of subsequent packets. ADAPTIVE performs significantly better than HASH, improving tail latency by 5× at 1.8 Gbps. We expect that implementing such a load balancing scheme in hardware would not be as straightforward; this example demonstrates the programmability of BESS.

6.2 Research Projects Based on BESS

The premise of BESS is to provide a highly flexible and modular framework for network dataplane, without compromising on performance. Because of its flexibility, BESS can be an effective building block for supporting and enhancing emerging networked systems. Many research and commercial projects have been built on BESS, by ourselves and colleagues in academia and industry. This section illustrates two BESS-derived research projects [88, 135]

that the author of this dissertation participated in as a coauthor.

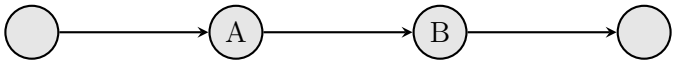
6.2.1 E2: A Framework for NFV Applications

Inspired by the benefits of cloud computing, network function virtualization (NFV) aims to bring the greater agility and efficiency of software to the processing of network traffic. While NFV has quickly gained significant momentum in both industry and academia alike, a closer look under the hood reveals a less rosy picture. More specifically, the following challenges must be addressed first to fully realize the promise of NFV:

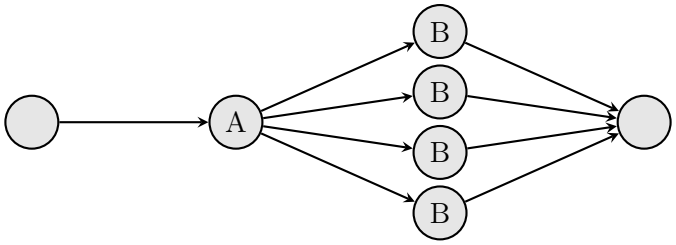
- Orchestration: how to integrate individual NFs to compose a "service chain"?
- Automation: how to automate common management tasks, such as load balancing or monitoring, for various types of NFs?
- Isolation: how to multiplex multiple traffic streams (from different users, or "tenants"), while ensuring security, protection, and fair use of resources among them?
- Scalability: how to scale in/out the system elastically in response to load changes, without disrupting quality of service?
- Utilization: how to maximize the hardware resource utilization of the system?
- Performance: how to avoid throughput/latency overheads associated with virtualization?

E2 [88] is our proof-of-concept research system for NFV orchestration. E2 is our attempt to develop novel, NF-agnostic methods to build a scalable NFV runtime system. E2 addresses various common issues in NFV deployment, including: placement,elastic scaling, service composition, resource isolation, and fault tolerance. We highlight two notable schemes we devised and implemented in E2. First, we developed a scheduling algorithm (which NFs should be placed where) that allows efficient resource utilization. When multiple NFs form a compound service, individual NF instances should be placed in a way that minimizes inter-server traffic, in order to save processor cycles and network bandwidth. Our scheme models NF instances and their connectivity as a graph partitioning problem (NP-hard) and solves it with a heuristic loosely based on the Kernighan-Lin algorithm. Second, we introduced a strategy called migration avoidance. This strategy supports dynamic scaling of stateful NFs without breaking "affinity", where traffic for a given flow must reach the instance that holds the state of the flow. Our scheme improves previous work with two benefits: 1) it supports legacy NFs without any state migration support and 2) it minimizes resource usage in hardware switches.

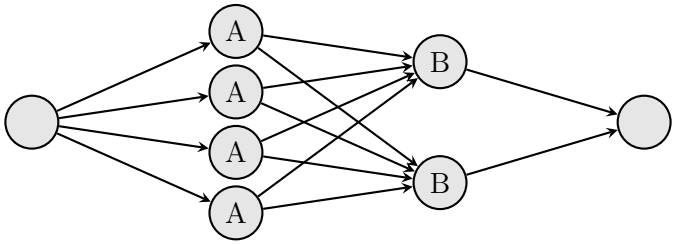
The E2 central controller takes a high-level "policy" graph, which is a chain of network functions that captures how network traffic should be processed by NFs in a declarative manner. Based on the policy graph, the controller makes scheduling decisions, such as how



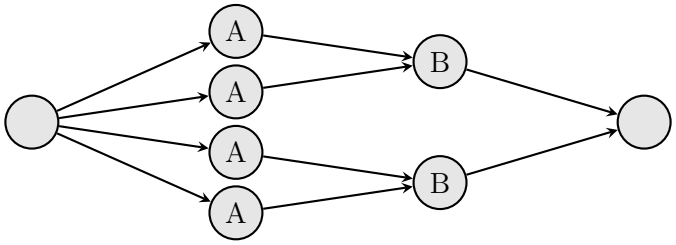
(a) Original policy graph, a chain of NF A and B



(b) Instance graph with split NF B



(c) Instance graph with split NF A and B



(d) Optimized instance graph

Figure 6.3: Transformations of a high-level policy graph (a) into an instance graph (b, c, d).

many physical NF instances should be deployed and what servers the NF instances should be assigned to, as illustrated in Figure 6.3. (a) shows an example of policy graph, which consist of a source node, logical NF nodes A and B, and a sink node. This graph represents that incoming packets are processed by NF A first, then by B. (b) is an example "instance" graph, where NF B requires four physical instances to handle the load. This NF split requires distributing the input traffic (after NF A) across all instances. Similarly, in (c), NF A needs four instances, while B needs two. In this case, the mesh between NF A instances and B instances represents that output packets from any NF A instance may be distributed to any instance of NF B. Assuming two servers, (d) depicts an optimal instantiation of the original policy graph; by placing two A instances and one B instance on each server, we can eliminate the inter-server traffic between NF A and B.

Once physical NF instances deployed across servers, they must be interconnected in a way that certain external traffic is classified and processed along the service chain according to the original policy graph. BESS is a central building block of the E2 dataplane, running on each server in the E2 cluster. BESS processes all packets coming in and out from the virtual network function instances running on the server as a gateway. The packet processing operations done by BESS include 1) classification: determining which policy graph packets belong to and what the next action should be, 2) load balancing: distributing packets across physical NF instances, 3) routing: redirecting packets if they are supposed to be processed in another server, and 4) inline NF processing: simple network functions, such as NAT and stateless ACL functions, can be directly implemented as BESS modules and embedded in the dataflow graph, to eliminate the packet I/O overhead imposed by VM/container virtual ports.

For the E2 dataplane implementation, BESS was an ideal platform for multiple reasons. First, since BESS is designed to be highly modular, any changes in the E2 dataplane for experimental features could be easily tested and verified by combining a few new and existing BESS modules, without requiring heavy modification in BESS system itself. This flexibility is not feasible with other purpose-built virtual switches, whose features are statically hardwired for particular functionality. Second, the BESS pipeline can be configured in a highly streamlined manner for optimal performance. Sources of packet processing overheads, such as unnecessary packet checks or operations or support for unused protocols, can be completely avoided if the E2 dataplane do not need them.

6.2.2 S6: Elastic Scaling of Stateful Network Functions

Elastic scaling, which is one of the most important benefits of NFV, is the ability to increase or decrease the number of physical NF instances for a logical NF, in order to adapt to changes in offered load. However, realizing elastic scaling has proven challenging due to the fact that most NFs are stateful; NFs instances must maintain shared state, which may be read or updated very frequently. For such stateful NFs, merely spinning up new physical NF instances and send some portion of the traffic to them is not sufficient to guarantee correct NF behavior. Elastic scaling must support access to shared state in a manner that

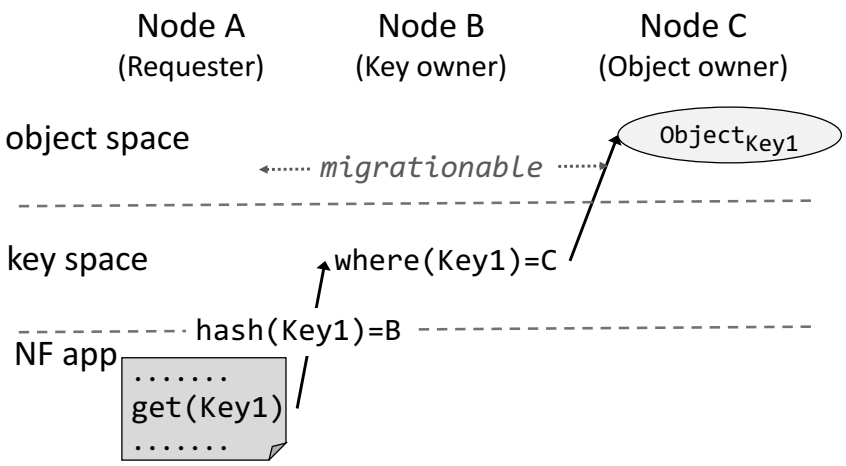


Figure 6.4: Distributed shared object (DSO) space, where object distribution is done in two layers: key and object layers. The key layer partitions the key space; each node keeps track of the current location of every object in the key partition. The object layer stores the actual binary of objects. The key layer provides indirect access to objects, allowing great flexibility in object placement.

ensures the consistency requirements of that state are met, and at the same time, without sacrificing NF throughput and latency. Previous work on scaling out of stateful NF have shown limitations in performance (*e.g.,* high packet processing latency due to remote access or data migration), functionality (*e.g.,* limited operation support for shared state), and/or development complexity (*e.g.,* requiring heavy modification to the design of existing NF implementations).

S6 [135] is our NF development and runtime framework, to support elastic scaling of NFs without compromising performance. Its design builds on the insight that a distributed shared state abstraction is well-suited to the NFV context. As depicted in Figure 6.4, S6 organizes state as a distributed shared object (DSO) space and extends the DSO concept with techniques designed to meet the need for elasticity and high-performance in NFV workloads. S6 extends the traditional DSO scheme as follows: 1) S6 dynamically reorganizes the DSO keyspace for space elasticity; 2) upon scaling events, state migrates in a "lazy" manner to minimize the downtime; 3) S6 utilizes per-packet microthreads to hide the remote access latency transparently and; 4) S6 extends the C++ syntax, so that developers can give S6 "hints" on migration and caching strategies on a per-object basis. Each object hence can have different consistency guarantees depending on their usage, enabling per-object optimization with trade-off between consistency requirements and performance. S6 simplifies NF development by hiding all the internal complexities of distributed state management under the hood, allowing developers to focus on NF-specific application logic.

In our prototype implementation of S6, BESS BESS was used as a dataplane virtual switch. NF instances built on S6 run in Linux containers, and BESS running in the host interconnects the containers with the physical interfaces. In addition to providing basic con-

nectivity to S6 containers, BESS performs two additional roles for the S6 system evaluation. The extreme flexibility of BESS enables us to evaluate S6 in various what-if test scenarios quickly and easily, without requiring any specialized hardware.

- **Traffic load balancing.** For evaluation of S6, we use BESS as a traffic load balancer to distribute incoming packets across S6 containers. The load balancing scheme is highly configurable (*e.g.,* what packet header fields should be used for hash-based packet distribution), and can be dynamically programmed at runtime. While S6 itself does not require any specific behaviors or guarantees from the load balancer in the cluster—it only assumes that the network somehow distributes input traffic across S6 instances—the programmability of BESS allows us to evaluate the effectiveness of S6 under various conditions. For example, we can test how well S6 behaves if input load is imbalanced among S6 instances. Also, we were able to test the worst-case scenario in terms of state access pattern, by artificially distributing packets in a way that state has to migrate without exhibiting any data locality.

- **Traffic generation.** We also use BESS as a traffic generator to measure the performance of the S6 cluster. BESS is configured to generate input packets to simulate various traffic characteristics, such as number of concurrent flows, the rate of new flows, the pattern of packet arrival, and the overall packet rate. BESS can easily achieve this thanks to its built-in scheduler with fine-grained resource usage control over a massive number of traffic classes. Also, BESS receives the return traffic processed by NF instances and measures throughput, drop date, latency, and jitter on a per-flow basis. Typically, these functionalities are provided by dedicated, expensive hardware traffic generator appliances. Unlike purpose-built virtual switches, the flexibility of BESS enabled easy addition of traffic generation and measurement features.

Chapter 7

Related Work

7.1 MegaPipe

A variety of work shares similar goals and/or approaches with MegaPipe in many different contexts. Fundamentally, there are two different philosophies: incrementally improving current solutions, or electing to take the clean-slate approach. As a clean-slate approach with a fresh API, MegaPipe does require slight modifications to applications, but we think these changes are justifiable given the performance improvements MegaPipe achieves, as shown in §4. In this section, we explore the similarities and differences between MegaPipe and existing work.

7.1.1 Scaling with Concurrency

Stateless event multiplexing APIs, such as `select()` or `poll()`, scale poorly as the number of concurrent connections grows since applications must declare the entire *interest set* of file descriptors to the kernel repeatedly. Banga et al. address this issue by introducing stateful interest sets with incremental updates [10], and we follow the same approach in this work with `mp_(un)register()`. The idea was realized with with `epoll` [30] in Linux (also used as the baseline in our evaluation) and `kqueue` [68] in FreeBSD. Note that this scalability issue in event delivery is orthogonal to the other scalability issue in the kernel: VFS overhead, which is addressed by lwsocket in MegaPipe.

7.1.2 Asynchronous I/O

Like MegaPipe, Lazy Asynchronous I/O (LAIO) [27] provides an interface with completion notifications, based on "continuation". LAIO achieves low overhead by exploiting the fact that most I/O operations do not block. MegaPipe adopts this idea, by processing non-blocking I/O operations immediately as explained in §5.2.

POSIX AIO defines functions for asynchronous I/O in UNIX [124]. POSIX AIO is not particularly designed for sockets, but rather, general files. For instance, it does not

have an equivalent of `accept()` or `shutdown()`. Interestingly, it also supports a form of I/O batching: `lio_listio()` for AIO commands and `aio_suspend()` for their completion notifications. This batching must be explicitly arranged by programmers, while MegaPipe supports transparent batching.

Event Completion Framework [31] in Solaris and `kqueue` [68] in BSD expose similar interfaces (completion notification through a completion port) to MegaPipe (through a channel), when they are used in conjunction with POSIX AIO. These APIs associate individual AIO operations, not handles, with a channel to be notified. In contrast, a MegaPipe handle is a member of a particular channel for explicit partitioning between CPU cores. Windows IOCP [134] also has the concept of completion port and membership of handles. In IOCP, I/O commands are not batched, and handles are still shared by all CPU cores, rather than partitioned as lwsockets.

7.1.3 System Call Batching

While MegaPipe's batching was inspired by FlexSC [118, 119], the main focus of MegaPipe is I/O, not general system calls. FlexSC batches synchronous system call requests via asynchronous channels (syscall pages), while MegaPipe batches asynchronous I/O requests via synchronous channels (with traditional exception-based system calls). Loose coupling between system call invocation and its execution in FlexSC may lead poor cache locality on multi-core systems; for example, the `send()` system call invoked from one core may be executed on another, inducing expensive cache migration during the copy of the message buffer from user to kernel space. Compared with FlexSC, MegaPipe explicitly partitions cores to make sure that all processing of a flow is contained within a single core.

netmap [103] extensively use batching to amortize the cost of system calls, for high-performance, user-level packet I/O. MegaPipe follows the same approach, but its focus is generic I/O rather than raw sockets for low-level packet I/O.

7.1.4 Kernel-Level Network Applications

Some network applications are partly implemented in the kernel, tightly coupling performance-critical sections to the TCP/IP stack [57]. While this improves performance, it comes at a price of limited security, reliability, programmability, and portability. MegaPipe gives user applications lightweight mechanisms to interact with the TCP/IP stack for similar performance advantages, while retaining the benefits of user-level programming.

7.1.5 Multi-Core Scalability

Past research has shown that partitioning cores is critical for linear scalability of network I/O on multi-core systems [19, 20, 91, 128]. The main ideas are to maintain flow affinity and minimize unnecessary sharing between cores. In §3.3.4, we addressed the similarities and differences between Affinity-Accept [91] and MegaPipe. In [20], the authors address the

scalability issues in VFS, namely inode and dentry, in the general context. We showed in §3.3.4 that the VFS overhead can be completely bypassed for network sockets in most cases.

The Chronos [59] work explores the case of direct coupling between NIC queues and application threads, in the context of multi-queue NIC and multi-core CPU environments. Unlike MegaPipe, Chronos bypasses the kernel, exposing NIC queues directly to user-space memory. While this does avoid in-kernel latency/scalability issues, it also loses the generality of TCP connection handling which is traditionally provided by the kernel.

7.1.6 Similarities in Abstraction

Common Communication Interface (CCI) [8] defines a portable interface to support various transports and network technologies, such as Infiniband and Cray's Gemini. While CCI and MegaPipe have different contexts in mind (user-level message-passing in HPC vs. general sockets via the kernel network stack), both have very similar interfaces. For example, CCI provides the *endpoint* abstraction as a channel between a virtual network instance and an application. Asynchronous I/O commands and notifications are passed through the channel with similar API semantics (e.g., `cci_get_event()`/`cci_send()` corresponding to `mp_dispatch()`/`mp_write()`).

The channel abstraction of MegaPipe shares some similarities with Mach port [2] and other IPC mechanisms in microkernel designs, as it forms queues for typed messages (I/O commands and notifications in MegaPipe) between subsystems. Especially, Barrelfish [11] leverages message passing (rather than sharing) based on event-driven programming model to solve scalability issues, while its focus is mostly on inter-core communication rather than strict intra-core communication in MegaPipe.

7.2 BESS

7.2.1 Click

The modular packet processing pipeline of BESS is inspired by the Click modular router [65]. We briefly discuss how we adapt Click's design for BESS. In general, Click's elements are defined in a much more fine-grained manner, *e.g.,* the `Strip` element removes a specified number of bytes from the packet, while BESS modules embody entire NIC functions (*e.g.,* switching is a single module). We chose to go with relatively coarse-grained modules because frequent transitions among modules has a significant impact on performance—small, dynamic modules provide compilers with limited optimization opportunities.

Furthermore, in BESS we assume that each module internally implements its own queues as necessary, whereas Click's scheduling is centered around providing explicit queues in the pipeline. This design choice simplifies our scheduling. The absence of explicit queues greatly streamlines packet processing in BESS; it can simply pick a traffic class and process packets using run to completion, without having to deal with "push" and "poll" calls as in Click. Another advantage of forgoing explicit queues is scalability. For example, in Click, supporting

1,000 rate limiters requires there be 1,000 explicit queues and token bucket elements in the dataflow graph, requiring the scheduler to consider all of these elements individually. In contrast, with the rate limiter of BESS (§6.1.2), the scheduler only needs to pick a traffic class to serve, simplifying the decision making.

For CPU scheduling, Click executes elements with weighted round-robin, enforcing fairness in terms of the number of executions. This is not a meaningful measure for either bandwidth (packet sizes differ) or processor time (packets may consume vastly different CPU cycles). We extend this scheduling model with explicit traffic class support, fixed priority, hierarchical composition, and accurate resource usage accounting. These extensions enable BESS to provide high-level performance isolation and guarantees across applications.

7.2.2 High-performance packet I/O

Efficient packet I/O is one of the most important enabling function of BESS, to implement its "fast-path" packet processing pipeline. BESS heavily relies on the recent advances in this area [24, 34, 41, 48, 103], such as batched packet I/O to save CPU cycles and reduce the cost of PCIe access, minimization of per-packet overhead on critical path, use of hugepages, lightweight packet buffers, busy-wait polling on dedicated cores, and bypassing operating system kernel.

7.2.3 Hardware support for virtual network I/O

The low performance of software-based virtual network I/O has received much attention [78, 107, 109, 129]. IOMMU-enabled servers allows VM guests to access the NIC hardware directly [15], to bypass the hypervisor network stack without compromising security. On top of IOMMU, NICs with the SR-IOV feature expose the hardware slices to guests, allowing them to share the same NIC [25]. While this hypervisor-bypass approach can achieve near native performance, its applicability is quite limited, especially in the context of multi-tenant datacenters, due to the rudimentary switching capability [99] and restricted means of traffic policy enforcement (*e.g.,* rate limiting [114], tunneling [66], and security filtering) of commodity NIC hardware.

We showed this trade-off in virtual network I/O between flexibility and performance is not fundamental. Note that many previous research papers have presumed that hardware-based NIC virtualization is the necessary norm and urged NIC vendors to implement their specific requirements (e.g., [60, 93, 96, 114]), but we argued that it is very unlikely to be timely. Our results indicate that software-based virtual network I/O can achieve both flexibility and performance, with careful design and implementation of software fast-path.

7.2.4 Smart NICs

Given the central role of networking in datacenter computing, the capability of NICs (especially in the virtualization context) have recently drawn significant attention. Many

previous research projects (e.g., sNICh [100], BMGNIC [82], FasTrak [96], Arrakis [93]) have suggested to augment some specific features to hardware NICs for additional functionalities or higher performance. We showed that BESS can implement many sophisticated features in software with little performance impact, even if we assume minimal hardware supports.

Concerning flexibility, there are commercial/research NICs that incorporate low-power processors or FPGA, which thus can be used as programmable NICs [6, 28, 70, 110, 120]. BESS on general-purpose processors has several advantages over those programmable hardware NICs: higher visibility and controllability without vendor/model specifics, elasticity in computation power provisioning, and easier development.

7.2.5 Ideal hardware NIC model

Although BESS can coexist with existing hardware NICs, an important benefit of the modular design is that BESS could act as an enabler for improved NIC designs in the long term. While there is growing consensus on exploiting NIC resources [115, 116], the interface and implementation of *black-box* (*e.g.,* "do checksum for this SCTP packet") features of recent NIC hardware present difficulties. We argue that NICs should instead offer *components* (*e.g.,* "calculate checksum for this payload range with CRC32 and update the result here"), so that BESS can offload a subset of dataflow graph to the hardware NIC, by compounding small yet reusable hardware building blocks into complete features. In this way, NICs can provide fine-grained behavior control, better state visibility, future proofness. We expect that the P4 [18] work can be a good starting point for this ideal NIC model.

7.2.6 Alternative approaches

BESS has several benefits over other approaches. Implementing network dataplane in legacy network stacks is slow, which is why hardware approaches gained popularity in the first place. Another alternative is to make the hardware more programmable (*e.g.,* FPGA [70, 110] and network processors [111]). However, the degree of programmability is still limited as compared to software on commodity servers, and hardware resource limitations still apply. Having a general-purpose processor on the NIC card, as a bump-in-the-wire, would be another option that is similar in spirit to BESS. We argue that BESS's approach (reusing existing server resources) is superior, in terms of efficiency/elasticity of resources and maintaining a global view of system.

Chapter 8

Concluding Remarks

In this dissertation we have presented two systems, MegaPipe (Chapter 3) and BESS (Chapter 5). The former targets at a more traditional application of network software: network server applications running on endhosts. Server applications utilize the socket abstraction, rather than directly dealing with raw packets, to communicate with clients. The latter focuses on a recently emerging application: software-based network functions running in-network. Network functions process network traffic at the packet level. Overall, MegaPipe and BESS represent higher-layer and lower-layer of the network stack respectively. One interesting avenue for future work is vertical integration of MegaPipe and BESS, so that the network stack can be truly programmable across all the layers, while providing optimal performance comparable to specialized, fixed-function network stack implementations.

Going forward, we believe that the techniques developed in the both systems as shown in this dissertation would be a practical guide to building software packet processing systems. The source code of MegaPipe and BESS have been publicly released for the academic and industry communities as an open source project under a BSD license. We expect that they can be a useful foundation for both researchers and systems developers to build upon.

Bibliography

[1] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. "Symbiotic Routing in Future Data Centers". In: *ACM SIGCOMM*. 2010.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation For UNIX Development". In: *Proceedings of USENIX Summer*. 1986.

[3] Brian Aker. *memaslap: Load testing and benchmarking a server*. http://docs.libmemcached.org/memaslap.html. 2012.

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, 2012.

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. "pFabric: Minimal Near-Optimal Datacenter Transport". In: *ACM SIGCOMM*. 2013.

[6] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware". In: *ACM SIGCOMM*. 2010.

[7] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Second. Prentice Hall, 1996.

[8] Scott Atchley, David Dillow, Galen Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. "The Common Communication Interface (CCI)". In: *Proceedings of IEEE HOTI*. 2011.

[9] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards Predictable Datacenter Networks". In: *ACM SIGCOMM*. 2011.

[10] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. "A Scalable and Explicit Event Delivery Mechanism for UNIX". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 1999.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The Multikernel: A new OS architecture for scalable multicore systems". In: *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[13] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services". In: *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 2003.

[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: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[15] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. "The Price of Safety: Evaluating IOMMU Performance". In: *Ottawa Linux Symposium*. 2007.

[16] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. "Understanding data center traffic characteristics". In: *ACM SIGCOMM Computer Communication Review (CCR)* 40.1 (2010), pp. 92–99.

[17] Denis Bilenko. *gevent: A coroutine-based network library for Python*. http://www.gevent.org/.

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

[19] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. "Corey: An Operating System for Many Cores". In: *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2008.

[20] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. "An Analysis of Linux Scalability to Many Cores". In: *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[21] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. "Non-scalable locks are dangerous". In: *Proceedings of the Linux Symposium*. 2012.

[22] B. Briscoe. *Tunnelling of Explicit Congestion Notification*. RFC 6040. Nov. 2010, pp. 1–35. URL: http://tools.ietf.org/html/rfc6040.

[23] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. "CPU DB: Recording Microprocessor History". In: *Commun. ACM* 55.4 (Apr. 2012), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2133806.2133822. URL: http://doi.acm.org/10.1145/2133806.2133822.

[24] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. "RouteBricks: Exploiting Parallelism to Scale Software Routers". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[25] Yaozu Dong, Xiaowei Yang, Li Xiaoyong, Jianhui Li, Haibin Guan, and Kun Tian. "High Performance Network Virtualization with SR-IOV". In: *IEEE HPCA*. 2012.

[26] Norbert Egi, Mihai Dobrescu, Jianqing Du, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, Laurent Mathy, et al. *Understanding the Packet Processing Capability of Multi-Core Servers*. Tech. rep. LABOS-REPORT-2009-001. EPFL, Switzerland, Feb. 2009.

[27] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. "Lazy Asynchronous I/O For Event-Driven Servers". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2004.

[28] *Emulex*. http://www.emulex.com.

[29] Ralf S. Engelschall. "Portable Multithreading - The Signal Stack Trick for User-Space Thread Creation". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2000.

[30] *epoll - I/O event notification facility*. http://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html. 2010.

[31] *Event Completion Framework for Solaris*. http://developers.sun.com/solaris/articles/event_completion.html.

[32] Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, and Igor Gashinsky. "Overclocking the Yahoo! CDN for Faster Web Page Loads". In: *Proceedings of ACM IMC*. 2011.

[33] Mario Flajslik and Mendel Rosenblum. "Network Interface Design for Low Latency Request-Response Protocols". In: *USENIX Annual Technical Conference (ATC)*. 2013.

[34] Francesco Fusco and Luca Deri. "High Speed Network Traffic Analysis with Commodity Multi-Core Systems". In: *ACM IMC*. 2010.

[35] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network requirements for resource disaggregation". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 249–264.

[36] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. "ELI: Bare-Metal Performance for I/O Virtualization". In: *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2012.

[37] J. Gross, T. Sridhar, P. Garg, C. Wright, and I. Ganga. *Geneve: Generic Network Virtualization Encapsulation*. IETF draft, http://tools.ietf.org/html/draft-gross-geneve-00.

[38] *gRPC: A high performance, open-source universal RPC framework*. https://grpc.io, retreived 05/01/2018.

[39] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. "Network support for resource disaggregation in next-generation datacenters". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM. 2013, p. 10.

[40] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. EECS Department, University of California, Berkeley, May 2015.

[41] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. "PacketShader: a GPU-Accelerated Software Router". In: *ACM SIGCOMM*. 2010.

[42] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[43] Sangjin Han and Sylvia Ratnasamy. "Large-Scale Computation Not at the Cost of Expressiveness". In: *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. 2013.

[44] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. "Efficient and Scalable Paravirtual I/O System". In: *USENIX Annual Technical Conference (ATC)*. 2013.

[45] Tom Herbert. *RPS: Receive Packet Steering*. http://lwn.net/Articles/361440/. 2009.

[46] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. "Is It Still Possible to Extend TCP?" In: *ACM IMC*. 2011.

[47] Bert Hubert. *Linux Advanced Routing and Traffic Control*. http://www.lartc.org.

[48] Intel. *Data Plane Development Kit (DPDK)*. http://dpdk.org.

[49] *Intel 10 Gigabit Ethernet Adapter*. http://e1000.sourceforge.net/.

[50] *Intel 8259x 10G Ethernet Controller*. Intel 82599 10 GbE Controller Datasheet. 2009.

[51] *Intel 8259x 10G Ethernet Controller*. Intel 82599 10 GbE Controller Datasheet. 2009.

[52] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS operating systems review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[53] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. *Silo: Predictable Message Completion Time in the Cloud*. Tech. rep. MSR-TR-2013-95. Sept. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=201418.

[54] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[55] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility". In: *ACM SIGCOMM*. 2014.

[56] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. "EyeQ: Practical network performance isolation at the edge". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.

[57] Philippe Joubert, Robert B. King, Rich Neves, Mark Russinovich, and John M. Tracey. "High-Performance Memory-Based Web Servers: Kernel and User-Space Performance". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2001.

[58] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. "Raising the Bar for Using GPUs in Software Packet Processing". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.

[59] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Vahdat Amin. *Reducing Datacenter Application Latency with Endhost NIC Support*. Tech. rep. CS2012-0977. UCSD, 2012.

[60] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. "Chronos: Predictable Low Latency for Data Center Applications". In: *ACM Symposium on Cloud Computing (SoCC)*. 2012.

[61] J. Kempf and R. Austein. *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*. RFC 3724. Mar. 2004. URL: http://tools.ietf.org/html/rfc3724.

[62] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. "The Power of Batching in the Click Modular Router". In: *ACM APSys*. 2012.

[63] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. "NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors". In: *ACM European Conference on Computer Systems(EuroSys)*. 2015.

[64] S.R. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX". In: *Proceedings of USENIX Summer*. 1986.

[65] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click modular router". In: 18.3 (Aug. 2000), pp. 263–297.

[66] Teemu Koponen et al. "Network Virtualization in Multi-tenant Datacenters". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[67] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. "Events Can Make Sense". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2007.

[68] Jonathan Lemon. "Kqueue: A generic and scalable event notification facility". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2001.

[69] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[70] 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 MSE*. 2007.

[71] Cameron Macdonell. "Shared-Memory Optimizations for Virtual Machines". PhD thesis. University of Alberta, 2011.

[72] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. IETF draft, http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-09.

[73] Ilias Marinos, Robert NM Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *ACM SIGCOMM*. 2014.

[74] Ilias Marinos, Robert NM Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *ACM SIGCOMM*. 2014.

[75] Nick Mathewson and Niels Provos. *libevent - an event notification library*. http://libevent.org.

[76] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *ACM SIGCOMM Computer Communication Review (CCR)* 38.2 (2008), pp. 69–74.

[77] *memcached - a distributed memory object caching system*. http://memcached.org/.

[78] Aravind Menon, Alan L Cox, and Willy Zwaenepoel. "Optimizing Network Virtualization in Xen". In: *USENIX Annual Technical Conference (ATC)*. 2006.

[79] Maged M Michael and Michael L Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. In: *ACM PODC*. 1996.

[80] Bosko Milekic. “Network Buffer Allocation in the FreeBSD Operating System”. In: *BSDCAN*. 2004.

[81] Jeffrey C Mogul. “TCP offload is a dumb idea whose time has come”. In: *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 2013.

[82] Jeffrey C. Mogul, Jayaram Mudigonda, Jose Renato Santos, and Yoshio Turner. “The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds”. In: *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 2013.

[83] JEFFREY C MOGUL and KK RAMAKRISHNAN. “Eliminating Receive Livelock in an Interrupt-Driven Kernel”. In: *ACM Transactions on Computer Systems (TOCS)* 15.3 (1997), pp. 217–252.

[84] *Netperf network benchmark software*. http://netperf.org.

[85] *Node.js: an event-driven I/O server-side JavaScript environment*. http://nodejs.org.

[86] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Stutsman Ryan. “The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM”. In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 92–105.

[87] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. “Flash: An Efficient and Portable Web Server”. In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 1999.

[88] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. “E2: A Framework for NFV Applications”. In: *Symposium on Operating Systems Principles (SOSP)*. 2015.

[89] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. “NetBricks: Taking the V out of {NFV}”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 203–216.

[90] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. “Fastpass: A Centralized "Zero-queue" Datacenter Network”. In: *ACM SIGCOMM*. 2014.

[91] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. “Improving Network Connection Locality on Multicore Systems”. In: *Proceedings of ACM European Conference on Computer Systems(EuroSys)*. 2012.

[92] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. “Improving Network Connection Locality on Multicore Systems”. In: *Proceedings of ACM European Conference on Computer Systems(EuroSys)*. 2012.

[93] 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*. Tech. rep. UW-CSE-13-10-01. University of Washington, May 2014.

[94] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. "The Design and Implementation of Open vSwitch". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.

[95] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, and Yoshio Turner Jose Renato Santos. "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing". In: *ACM SIGCOMM*. 2013.

[96] Radhika Niranjan Mysore George Porter and Amin Vahdat. "FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers". In: *ACM CoNEXT*. 2013.

[97] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. "SENIC: Scalable NIC for End-Host Rate Limiting". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014.

[98] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, Mark Handley, et al. "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[99] Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, Scott Rixner, Thomas W Barr, Rebecca Smith, and Scott Rixner. "Hyper-Switch: A Scalable Software Virtual Switching Architecture". In: *USENIX Annual Technical Conference (ATC)*. 2013.

[100] Kaushik Kumar Ram, Jayaram Mudigonda, Alan L Cox, Scott Rixner, Parthasarathy Ranganathan, and Jose Renato Santos. "sNICh: Efficient Last Hop Networking in the Data Center". In: *ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*. 2010.

[101] K. Ramakrishnan, S. Floyd, and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001, pp. 1–62. URL: http://tools.ietf.org/html/rfc3168.

[102] *Receive-Side Scaling*. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx. 2008.

[103] Luigi Rizzo. "netmap: a novel framework for fast packet I/O". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2012.

[104] Luigi Rizzo and Giuseppe Lettieri. "VALE: a Switched Ethernet for Virtual Machines". In: *ACM CoNEXT*. 2012.

[105] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. "Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks". In: *USENIX WIOV*. 2011.

[106] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. "It's Time for Low Latency". In: *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 2011.

[107] Rusty Russell. "virtio: Towards a De-Facto Standard for Virtual I/O Devices". In: *ACM Operating Systems Review* 42.5 (2008), pp. 95–103.

[108] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design". In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

[109] Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Ian Pratt. "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization". In: *USENIX Annual Technical Conference (ATC)*. 2008.

[110] Jeffrey Shafer and Scott Rixner. "RiceNIC: A Reconfigurable Network Interface for Experimental Research and Education". In: *ACM ExpCS*. 2007.

[111] Niraj Shah. "Understanding Network Processors". MA thesis. University of California, Berkeley, 2001.

[112] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. "PISCES: A Programmable, Protocol-Independent Software Switch". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 525–538.

[113] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service". In: *SIGCOMM*. 2012.

[114] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. "Sharing the Data Center Network". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[115] Pravin Shinde, Antoine Kaufmann, Kornilios Kourtis, and Timothy Roscoe. "Modeling NICs with Unicorn". In: *ACM Workshop on Programming Languages and Operating Systems (PLOS)*. 2013.

[116] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. "We Need to Talk About NICs". In: *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 2013.

[117] *Snort Intrusion Detection System*. https://snort.org.

[118] Livio Soares and Michael Stumm. "Exception-Less System Calls for Event-Driven Servers". In: *Proceedings of USENIX Annual Technical Conference (ATC)*. 2011.

[119] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls". In: *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[120] *Solarflare Communications*. http://www.solarflare.com.

[121] *SPECweb2009 Release 1.20 Support Workload Design Document*. http://www.spec.org/web2009/docs/design/SupportDesign.html. 2010.

[122] M. Sridharan, A. Greenberg, Y. Wang, P. Gard, N.Venkataramiah, K. Duda, I. Ganga, G. Lin, M. Pearson, P. Thaler, and C. Tumuluri. *NVGRE: Network Virtualization using Generic Routing Encapsulation*. IETF draft, http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-04.

[123] Igor Sysoev. *nginx web server*. http://nginx.org/.

[124] *The Open Group Base Specifications Issue 7*. http://pubs.opengroup.org/onlinepubs/9699919799/. 2008.

[125] R Joshua Tobin and David Malone. "Hash Pile Ups: Using Collisions to Identify Unknown Hash Functions". In: *IEEE CRiSIS*. 2012.

[126] *Transactional Synchronization in Haswell*. https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, retreived 05/01/2018.

[127] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-chip Parallelism". In: *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*. ISCA '95. S. Margherita Ligure, Italy: ACM, 1995, pp. 392–403. ISBN: 0-89791-698-0. DOI: 10.1145/223982.224449. URL: http://doi.acm.org/10.1145/223982.224449.

[128] Bryan Veal and Annie Foong. "Performance Scalability of a Multi-Core Web Server". In: *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*. 2007.

[129] VMware. *Performance Evaluation of VMXNET3 Virtual Network Device*. http://http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf.

[130] *Vector Packet Processing*. https://wiki.fd.io/view/VPP, retreived 05/01/2018.

[131] Carl A Waldspurger and William E Weihl. *Stride Scheduling: Deterministic Proportional Share Resource Management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[132] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services". In: *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 2001.

[133] Paul Willmann, Scott Rixner, and Alan L Cox. "An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems". In: *USENIX ATC*. 2006.

[134] *Windows I/O Completion Ports*. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx. 2012.

[135] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. "Elastic scaling of stateful network functions". In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, pp. 299–312.

[136] Xerbert Xu. *Generic Segmentation Offload*. http://lwn.net/Articles/188489/.

[137] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012.