Title
Next Generation Datacenter Architecture

Author
Gao, Xiang

Publication Date
2018

Peer reviewed|Thesis/dissertation

Next Generation Datacenter Architecture

by

Xiang Gao

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, Co-chair
Professor Scott Shenker, Co-chair
Professor Rhonda Righter

Summer 2018

Next Generation Datacenter Architecture

Abstract

Next Generation Datacenter Architecture

by

Xiang Gao

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sylvia Ratnasamy, Co-chair

Professor Scott Shenker, Co-chair

Modern datacenters are the foundation of large scale Internet services, such as search engines, cloud computing and social networks. In this thesis, I will investigate the new challenges in building and managing large scale datacenters. Specifically, I will show trends and challenges in the software stack, the hardware stack and the network stack of modern datacenters, and propose new approaches to cope with these challenges.

In the software stack, there is a movement to serverless computing where cloud customers can write short-lived functions to run their workloads in the cloud without the hassle of managing servers. Yet the storage stack has not changed to accommodate serverless workloads. We build a storage system called Savanna that significantly improves the serverless applications performance. In the hardware stack, with the end of Moore's Law, researchers are proposing disaggregated datacenters with pools of standalone resource blades. These resource blades are directly connected by the network fabric, and I will present the requirements of building such a network fabric. Lastly, in the network stack, new congestion control algorithms are proposed (e.g. pFabric) to reduce the flow completion time. However, these algorithms require specialized hardware to achieve desirable performance. I designed pHost, a simple end-host based congestion control scheme that has comparable performance with pFabric without any specialized hardware support.

To my advisors, friends and my family who made this possible.

Contents

Contents **ii**

List of Figures **v**

List of Tables **viii**

1 Introduction **1**
1.1 New trends in modern datacenters 1
1.2 Architecting future datacenters 3
1.3 Contributions 4
1.4 Dissertation Plan 5

2 Background **6**
2.1 Light-weight Virtualization 6
2.2 Existing Disaggregation Prototypes 7
2.3 pFabric 8

3 An Architectural Extension for Serverless Computing **9**
3.1 Introduction 9
3.2 Background 11
3.2.1 Serverless Applications 11
3.2.2 New Serverless Workloads 11
3.2.3 Requirements from New Workloads 12
3.3 Savanna Design 13
3.3.1 Programming Model 14
3.3.2 Savanna Components 14
3.3.3 Consistency 16
3.3.4 Caching 17
3.3.5 Fault Tolerance 17
3.4 Implementation 18
3.5 Evaluation 19
3.5.1 Performance 20

3.5.2 Consistency . . . 25
3.5.3 Fault Tolerance . . . 26
3.5.4 Scalability . . . 27
3.5.5 Local Storage Constraints . . . 27
3.5.6 Case Study: Matrix Multiplication . . . 29
3.6 Related Work . . . 30
3.7 Conclusion . . . 31

4 Network Requirements for Resource Disaggregation 32
4.1 Introduction . . . 32
4.2 Disaggregated Datacenters . . . 34
4.2.1 Assumptions: Hardware Architecture . . . 35
4.2.2 Assumptions: System Architecture . . . 37
4.2.3 Design knobs . . . 38
4.3 Network Requirements . . . 38
4.3.1 Methodology . . . 39
4.3.2 Results . . . 41
4.3.3 Implications and Feasibility . . . 46
4.4 Network Designs for Disaggregation . . . 48
4.4.1 Methodology: DDC Traffic Workloads . . . 48
4.4.2 Methodology: Queuing delay . . . 49
4.4.3 Network-level performance . . . 50
4.4.4 Application-level performance . . . 51
4.5 Future Directions . . . 51
4.5.1 Implementing remote memory access . . . 52
4.5.2 Improved performance via disaggregation . . . 53
4.6 Related Work and Discussion . . . 54
4.7 Conclusion . . . 55

5 pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric 56
5.1 Introduction . . . 56
5.2 pHost Overview . . . 57
5.2.1 Modern Datacenter Networks . . . 57
5.2.2 Basic Transport Mechanism . . . 57
5.2.3 Why pHost works . . . 58
5.3 Design Details . . . 59
5.3.1 pHost Source and Destination . . . 59
5.3.2 Maximizing Network Utilization . . . 61
5.3.3 Local Scheduling Problem . . . 62
5.3.4 Handling Packet drops . . . 63
5.4 Evaluation . . . 64

5.4.1 Test Setup 64
5.4.2 Performance Evaluation Overview 67
5.4.3 Varying Metrics and Scenarios 67
5.4.4 Flexibility 74
5.5 Related Work 76
5.6 Conclusion 77

6 Future Work and Conclusion **78**

Bibliography **80**

List of Figures

2.1 Intel RSA Design 7

3.1 Savanna consists of a logically centralized metadata coordinator and a set of distributed Savanna agents, and relies on the cloud provider's blob store for durable storage. . . . 15
3.2 Savanna improves data analytics workloads and machine learning workloads by 3.3× to 6.4×. For IoT and traditional serverless workloads, it accelerates their speed between 1.4× and 3.0×. Savanna also improves the stability of the system as the runtime varies less across runs (smaller error bars). 21
3.3 Savanna reduces I/O time from 89%-94% to 38%-66% and it significantly outperforms PySpark and Databricks Serverless under the BDB_1 and Sort workloads. 22
3.4 Savanna reduces S3 usage. Therefore, both cloud customers and providers have motivation to deploy Savanna. 23
3.5 We evaluate YCSB A (50% writes and 50% reads) and YCSB B (5% writes and 95% reads) with varying file size. Savanna improves the I/O performance by 14× and 82× with local cache hits, and 5× to 40× with peer cache hits. Therefore, Savanna improves application performance on workloads with thousands of invocations (and spans multiple machines) because effective peer cache. 24
3.6 Disabling consistency only improves small files' performance by a little. For files larger than 1MB, disabling consistency introduces no benefit. In a typical cloud environment where file sizes are often larger than 1M, Savanna's consistency feature could be enabled by default. 25
3.7 In real workloads where file sizes are often large, there is no observable performance degradation when consistency feature is enabled. 26
3.8 The runtime of a no-failure run, a Savanna-recovery run, and resubmitting the entire job. The failures are triggered roughly half way through the executions, so resubmitting job after failure takes about 50% more time than the no-failure case. 26
3.9 Measuring Savanna's weak scaling performance by sorting 256GB, 512GB, 768GB and 1TB of 100 Byte records. The input data size to concurrent invocation ratio is kept at 1GB/invocation. 27
3.10 Measuring Savanna's strong scaling performance by sorting 512GB data using 32 to 512 concurrent invocations. 28

3.11 Runtime of sorting 160GB data with varying eviction ratio. Savanna's performance degrades gracefully under memory constraints. 29
3.12 Savanna has better scalability per core-hour. By using Savanna, the workload can be executed on smaller and cheaper instances, as Savanna reduces the data shuffle cost. . . 29

4.1 High-level architectural differences between server-centric and resource-disaggregated datacenters. 35
4.2 Comparison of application-level performance in disaggregated datacenters with respect to existing server-centric architectures for different latency/bandwidth configurations and 25% local memory on CPU blades — Class A apps (top) and Class B apps (bottom). To maintain application-level performance within reasonable performance bounds ($\sim$ 5% on an average), Class A apps require 5μs end-to-end latency and 40Gbps bandwidth, and Class B apps require 3μs end-to-end latency and $40-100$Gbps bandwidth. See §4.3.2 for detailed discussion. 39
4.3 Impact of network bandwidth on the results of Figure 4.2 for end-to-end latency fixed to 5μs and local memory fixed to 25%. 41
4.4 Impact of network latency on the results of Figure 4.2 for bandwidth fixed to 40Gbps and local memory fixed to 25%. 42
4.5 Impact of "local memory" on the results of Figure 4.2 for end-to-end latency fixed to 5μs and network bandwidth 40Gbps. Negative values are due to small variations in timings between runs. 42
4.6 Performance degradation of applications is correlated with the swap memory bandwidth and overall memory bandwidth utilization. 43
4.7 The performance of the five protocols for the case of 100Gbps access link capacity. The results for 40Gbps access links lead to similar conclusions. See §4.4.3 for discussion on these results. 49
4.8 Application layer slowdown for each of the four applications at rack-scale and datacenter scale after injecting pFabric's FCT with 100Gbps link. 52
4.9 Running COST in a simulated DDC. COST-DDC is 1.48 to 2.05 faster than GraphX-Server Centric except for one case. We use two datasets in our evaluation, UK-2007-05 (105m nodes, 3.7b edges), and Friendster (65m nodes, 1.8b edges) 53

5.1 In this example, two flows A and B arrive at the source at roughly the same time, and two RTS are sent to the respective destinations. Suppose the source gets a token for flow A first. Since the source has only one token, it immediately consumes it by sending the corresponding data packet to destination A. Now suppose the source receives another token for flow A and a token for flow B while it is sending the data packet for flow A. The source now has two tokens, one for each flow. Suppose the source decides to utilize the token for flow B at this step. 62
5.2 Distribution of flow sizes across workloads used in our evaluation. Note that short flows dominate all workloads; however, Data Mining and IMC10 workloads have significantly larger fraction of short flows when compared to the Web Search workload. . 65

5.3 Mean slowdown of pFabric, pHost, and Fastpass across different workloads for our default configuration (0.6 load, per-port buffers of 36kB). pHost performs comparable to pFabric, and 1.3–4× better than Fastpass. 66
5.4 Breakdown of mean slowdown by flow size for pFabric, pHost, and Fastpass (all flows greater than 10MB (for Data Mining and Web Search workloads) and greater than 100KB (for IMC10 workload) are considered long flows). All the three schemes have similar performance for long flows; for short flows, however, pHost performs similar to pFabric and 1.3–4× better than Fastpass. 66
5.5 Performance of the three protocols across various performance metrics. See §5.4.3 for detailed discussion. 69
5.6 Performance of the three protocols across varying network loads. 72
5.7 Stability analysis for pFabric. x-axis is the fraction of packets (out of total number of packets across the simulation) that have arrived at the source as the simulation progresses; y-axis is the fraction of packets (again, out of total number of packets across the simulation) that have yet not been injected into the network by the sources. pFabric is stable at 0.6 load, unstable beyond 0.7 load. We get similar results for pHost and Fastpass. 73
5.8 Mean slowdown of pHost, pFabric, and Fastpass in synthetic workload (with varying fraction of short flows). Both pFabric and pHost perform well when the trace is short flow dominated. Fastpass performs similar to pHost and pFabric when there are 90% long flows, but gets significantly worse as the fraction of short flows increases. 73
5.9 Performance of the three protocols across various traffic matrices. pHost performs better than pFabric and Fastpass for Permutation TM, and within 5% of pFabric for incast TM. 74
5.10 Both pHost and pFabric perform well even with tiny buffer sizes. Moreover, the performance of all the three protocols remains consistent across a wide range of switch buffer sizes. 75
5.11 pHost, by decoupling flow scheduling from the network fabric, makes is easy to implement diverse policy goals (*e.g.*, fairness in a multi-tenant scenario). In this figure, one tenant gets greater throughput with pFabric (for reasons discussed in §5.4.4), while the throughput is more fairly allocated using pHost. 75

List of Tables

3.1 Two types of named function restart 18
3.2 Workloads used in evaluation. Parallelism refers to the concurrent invocations per workload and our cluster has 16 slots per server. 20
3.3 Running the sort benchmark with different garbage collection scenarios. Only a GC that pauses `writes` impacts performance. 28

4.1 Typical latency and peak bandwidth requirements within a traditional server. Numbers vary between hardware. 35
4.2 Applications, workloads, systems and datasets used in our study. We stratify the classes in Section 4.3. 36
4.3 Class B apps require slightly higher local memory than Class A apps to achieve an average performance penalty under 5% for various latency-bandwidth configurations. . 41
4.4 Achievable round-trip latency (Total) and the components that contribute to the round-trip latency (see discussion in §4.3.3) on a network with 40Gbps access link bandwidth (one can further reduce the **Total** by 0.5μs using 100Gbps access link bandwidth). The baseline denotes the latency achievable with existing network technology. The fractional part in each cell is the latency for one traversal of the corresponding component and the integral part is the number of traversal performed in one round-trip time (see discussion in §4.3.3). 45
4.5 RDMA block device request latency(ns) 53

Acknowledgments

I would like to thank my advisors Sylvia Ratnasamy and Scott Shenker for their unlimited care and support throughout my PhD life. Both Sylvia and Scott are very smart and inspiring. They give enough freedom to me to develop my own ideas, and enough guidance so that I am always on track. As a non-native speaker, I need to work hard on paper writing and presentations. Sylvia and Scott are always available and give me extra attention when I need help. I remember the days and nights we spent on iterating my conference talks and paper submissions. Beyond academics, Sylvia and Scott demonstrate me how to be a person with enthusiasm, boldness and leadership. They are, and will continue to be my role models in the rest of my life. Lastly, I want to thank them for providing chocolates in 415 and the futon in 413 that survived me during paper deadlines.

I would like to thank my qualification exam and thesis committee Randy Katz and Rhonda Righter for their insightful discussions. Their comments made my thesis more sound and complete.

I would like to thank my collaborators who involved in my thesis work – Rachit Agarwal, Joao Carreira, Sangjin Han, Sagar Karandikar, Gautam Kumar, Akshay Narayan, Aurojit Panda, Ben Recht, Vaishaal Shankar, Qiyin Wu, Wen Zhang. Special thanks to Rachit Agarwal and Aurojit Panda who gave me hands-on guidance on system design and paper writing.

I would like to thank all my friends in NetSys Lab – Emmanuel Amaro, Soumya Basu, Christopher Canel, Michael Chang, Carlyn Chinen, Silvery Fu, Sangjin Han, Yotam Harchol, Anwar Hithnawi, Xiaohe Hu, Ethan Jackson, Keon Jang, Marc Korner, Gautam Kumar, Jon Kuroda, Chang Lan, Maziar Manesh, Murphy McCauley, Radhika Mittal, Aisha Mushtaq, Akshay Narayan, Kay Ousterhout, Shoumik Palkar, Aurojit Panda, Zafar Qazi, Luigi Rizzo, Torsten Runge, Colin Scott, Justine Sherry, Amin Tootoonchian, Zhang Wen, Shinae Woo, Jack Zhao. The discussions with you – on both research and life – are always fun and inspiring.

I would like to thank all my friends and professors in RISE Lab, ASPIRE Lab and the Firebox project – Krste Asanovic, Joao Carreira, Ali Ghodsi, Jenny Huang, Xin Jin, Sagar Karandikar, Randy Katz, Haoyuan Li, Howard Mao, Vaishaal Shankar, Vikram Sreekanti, Ion Stoica, Nathan Pemberton, Qifan Pu, Johann Smith, Liwen Sun, Reynold Xin.

Many thanks to Chi Ben, Kaifei Chen, Xihan Chen, Zhijie Chen, Yang Gao, Liang Gong, Chaoran Guo, Jin Chi, Peihan Miao, Shaxuan Shan, Hong Shang, Liwen Sun, Nan Tian, Di Wang, Jiannan Wang, Qiyin Wu, Renyuan Xu, Yang Yang, Hezheng Yin, Bodi Yuan, Kai Zeng, Ben Zhang, Zhao Zhang, Qian Zhong, Zemin Zhong, David Zhu and my other friends at Berkeley. You made my PhD life a wonderful experience.

Before joining Berkeley, I did my masters at the University of Waterloo. I want to show my appreciation to my old advisor Srinivasan Keshav, and collaborators Andy Curtis, James Liu, Daniel J. Lizotte, Rayman Preet Singh, Bernard Wong.

Lastly, I would like to thank my family including my parents, my girlfriend He Zhao for their support during my PhD. I could not do this without you.

Chapter 1

Introduction

By the end of 2017, roughly 4 billion people are connected by the Internet [8] and a large portion of Internet traffic goes to datacenters. With the continuous growth of Internet traffic towards datacenters, investment on datacenters will surge in the next few years. Cisco estimates that the number of hyperscale datacenters will grow from 338 (end of year 2016) to 628 in five years [4]. For Internet services, such as search engines, cloud computing platforms, online social networks, building high performance and low cost datacenters is the crux to their business. For example, Google is continuing expanding its datacenters globally [6]. Beyond building new datacenters, various cloud service providers are embracing new architectures [22, 50, 72] and technologies [47, 60, 69, 86] in their datacenters.

Modern datacenters are of large scale which introduces new challenges in terms of performance, cost, scaling, and management. In this thesis, we focus on several aspects of these challenges and propose multiple solutions to address them. First, while virtual machines provide strong performance isolation and security, the overhead incurred at the hypervisor layer is not negligible. This hinders the deployment of many applications in the cloud simply because of the performance overhead and the startup latency of virtual machines. Further, for users less experienced with cloud, setting up and managing a cloud environment is an extremely challenging task. This calls for a new software architecture with higher performance and lower barriers to entry. Second, Moore's Law has ended in the past few years and it becomes harder to scale within a single machine (i.e., building a CPU with many cores or with a high capacity memory controller). Thus, a more scalable and efficient new hardware stack is needed to overcome these new architecture constraints, such as the memory wall [139]. Lastly, with the growth of data intensive applications [51, 92, 142], there is a non-trivial bandwidth demand within datacenters. Effectively dealing with different forms of congestion [106, 138] without losing fairness is the new challenge.

1.1 New trends in modern datacenters

We observe multiple trends in the modern datacenters.

Serverless Computing. Virtual machines are inherently heavy weight and hard to manage. Recent advances in light weight container isolation [47, 86] provides an efficient alternative for virtualization. Instead of creating virtualized hardware, containers create virtualized operating systems, hence they are more light weight and simpler to deploy. With container isolation, software execution environment could be deployed with container images at extremely low latency. Compared to virtual machines which often take minutes to deploy, containers can be deployed in seconds or even subseconds. This enables serverless computing where cloud customers can submit function handlers to the cloud, and the cloud simply executes these functions in the containers. The containers not only provide a light weight execution environment, but also offer performance isolation between different cloud customers. With serverless computing, cloud customers do not need to install software stack on virtual machines, manage cluster scaling, and deal with fault tolerance. Instead, they can simply focus on implementing the business logic and offload the complexity of distributed computing to the cloud. Currently, all major cloud vendors have their own serverless offering [22, 24, 58] and there are many open source efforts on private serverless clouds [64, 100].

Disaggregated Datacenters. Modern datacenters are composed of servers, connected by the datacenter network fabric. However, the Moore's Law has ended and servers are becoming harder to scale. Therefore, driven by the hardware architecture community, a new disaggregated datacenter design is emerging. Instead of building servers with tightly coupled resources (i.e., CPU, memory, disk, NIC), a disaggregated datacenter is composed of resource blades that only contain one type of resource. This overcomes multiple architectural difficulties in terms of performance and power scaling. For example, the memory capacity wall [139] limits the capacity of on-board memory controllers – a high capacity memory controller incurs high access latency – making the tight integration of CPU and memory unsustainable. As a result, a disaggregated datacenter design would significantly simplify the motherboard design and enables individual server components to evolve independently. From the datacenter operators' perspective, disaggregated datacenter loosens the locality constraints between compute, storage and other components, hence offers new ways for datacenter resource provisioning and scheduling.

Priority-based Flow Scheduling. Datacenter fabric needs to handle a mixture of distinct workloads. These workloads often have various performance objectives. For example, for short flows generated by interactive queries and RPCs, achieving low latency is the major objective, while for large flows generated by data analytics applications, high throughput is more important. However, in a datacenter with a mixture of short flows and long flows, TCP will attempts to fairly allocate bandwidth to short flows and long flows. Even worse, because of TCP's slow start behavior and the high bandwidth delay product in modern datacenters, short flows finish before achieving their fair share. Therefore, in a datacenter, the ideal flow scheduling policy is to give priority to short flows so that they finish faster. At a first look, this policy seems to hurt fairness. However, a recent paper called pFabric [16] shows that giving higher priority to short flows improves short flow latency without hurting long flow performance. pFabric schedules flows by prioritizing those packets with the lowest remaining flow size at the switch queue. The simple scheduling scheme achieves near optimal performance in multiple datacenter workloads.

1.2 Architecting future datacenters

While these new trends could effectively overcome multiple limitations in previous datacenter designs, they introduce several new challenges. This thesis is an attempt to address the challenges by re-architecting the datacenter at the software stack, hardware stack and network stack.

Enhanced Serverless Computing. Serverless architecture is initially designed for short-lived event-based applications such as web serving and image processing. Recently, serverless computing has gain traction on applications such as data analytics, machine learning, and sensor data processing due to its management benefits. These emerging applications often have higher concurrency and throughput requirements. With more parallel threads running on the cloud, failures are common and fault-tolerance becomes a new requirement as well. With these new requirements, current serverless offerings fall short due to a lack of transaction service, cache service, and fault recovery service.

Although each of these individual services exists in the current cloud computing environment, using them requires extra expertise, which is a departure from serverless computing's ease of use philosophy. Further, the interaction between various services requires extra attention of the developers. For example, the programmer must hand over the locks from the transaction service to the recovery service so that the file locks can be successfully obtained on recovery run.

Instead of providing multiple APIs for different cloud services, we provide a unified and simplified interface to serverless users that encapsulates all the required services. More specifically, we propose Savanna that inserts a shim layer between the serverless applications and the storage. From the applications' perspective, Savanna is another storage layer, hence it is easy for programmers to use. From the storage's perspective, Savanna is just another application, so no API modification is needed. By unifying the transaction service, cache service, and fault recovery service, functions in Savanna can be abstracted as a single transaction, data objects in Savanna can be accessed faster, and data loss due to machine failure can be recovered automatically.

Network Requirements for Resource Disaggregation. As we have discussed, resource disaggregation has multiple benefits on hardware design, resource provisioning and scheduling. Therefore, many companies are working on disaggregated datacenter prototypes [72, 96, 126]. Since resource blades (i.e., CPU, memory and disk blades) are spread across the entire datacenter, the intra-server traffic (i.e., CPU-memory traffic) in server-centric datacenters is now inter-server traffic. Therefore, network fabric performance becomes the crux of disaggregated datacenters. To provide guaranteed high network performance, many existing disaggregation prototypes require specialized hardware, such as PCI-e network, silicon photonics, to connect the resource blades. While using specialized hardware assures performance, it could significantly increase the cost of the system. Therefore, we investigate if it is feasible to build disaggregated datacenters using *commodity* network equipment. We found that current commodity hardware is sufficient for resource disaggregation, but the current OS software stack is not. However, there are feasible solutions to overcome the performance limitations in the software stack.

Priority-based Congestion Control at Endhost. pFabric provides a promising mechanism for datacenter congestion control. However, pFabric relies on switches to implement priority queues – a feature that is not available in current switches. This makes pFabric hard to be deployed in datacenters.

To address this problem, we design pHost, a scheme that allows priority-based congestion control without specialized switches. pHost assumes full bisection bandwidth and this makes the design extremely simple – a flow receiver issues a *token* to a sender when it has free bandwidth, and a sender can only send a packet when it receives a *token* from a receiver recently. By sending a token, the receiver is reserving bandwidth for a short period to the packet sender. pHost can implement arbitrary flow scheduling policy by selecting which flow to issue token to (at the receiver) and which flow's token to consume (at the sender). For example, shortest remaining flow size scheduling can be achieved by selecting the currently shortest flow at both the sender and receiver side. This simple design can meet pFabric's performance without using commodity switches.

1.3 Contributions

This thesis investigates various aspects of current datacenter architecture – including software architecture, hardware architecture and network architecture – and proposes multiple solutions to address some problems. The contributions of the thesis include the following:

- The design and implementation of Savanna, an architecture that provides an unified API that encapsulates key cloud computing features such as consistency, caching, and fault tolerance. With Savanna, serverless programmers can simply use a file-like API to do I/O and achieve high performance. Our evaluation shows that Savanna improves application performance between 1.4× to 6.4×. Besides performance, Savanna guarantees per function snapshot isolation, a consistency model widely used in modern databases. On hardware failures, Savanna can automatically replay failed execution and can recover all lost states using the tracked lineage. Savanna is used by machine learning researchers at UC Berkeley and is open sourced for the community.
- A preliminary study on the network requirements for resource disaggregation. By creating a Linux kernel block device driver as a swap device, we emulate an environment where applications can use remote memory inside a datacenter. By placing bandwidth and latency constraints on the block device driver, we discover that roughly 100Gbps bandwidth and 3μs end-to-end latency are needed for the evaluated applications. We found that with network technologies such as RDMA and cut-through switching, modern datacenter hardware is sufficient to meet the aforementioned requirements. However, we do need to redesign the OS stack to reduce the software overhead. Lastly, we have open sourced our emulator for future researchers to evaluate on other workloads.
- The design of the pHost congestion control protocol. pHost has an extremely simple design that only requires changes at the endhosts. pHost can achieve near optimal flow completion

time performance without using specialized hardware. This makes priority-based congestion control practical in modern datacenters. Again, pHost is open sourced for future researches on datacenter congestion control.

1.4 Dissertation Plan

The rest of the dissertation is organized as follows. In Chapter 2, we describe in detail on current datacenter architecture. Chapter 3 describes the design and implementation of the Savanna system [56]. In Chapter 4, we demonstrate how we obtain the network requirements for resource disaggregation in current datacenters [54]. In Chapter 5, we present the pHost [55] design and evaluation. Finally, in Chapter 6, we discuss the future research directions and conclude the thesis.

Chapter 2

Background

We describe the current status quo in modern datacenters and how new technology trends are affecting the architecture of future datacenters.

2.1 Light-weight Virtualization

Previous virtualization technologies are based on virtual machines. In the Infrastructure as a Service (IaaS) cloud, cloud vendors rent virtual machines to the cloud customer. A virtual machine creates a virtualized hardware environment, and it provides isolation and security. The virtual machine software is often called hypervisor. Popular hypervisor software includes Xen [32], VMWare VSphere [1], Hyper-V [2]. Virtual machines enable safe and fair sharing of a physical machine to multiple users by creating virtual machines.

Container isolation [47, 86] is a new form of virtualization, where a container only virtualizes operating system rather than hardware. Containers are introduced because of the following new features in Linux kernel:

- **Kernel namespace** allows isolation between processes where processes in different namespaces see different sets of resources, such as CPU cores, partition mounts, network, interprocess communication. This provides the isolation feature for virtualization.
- **Cgroup** or control group allows fine grained resource sharing between processes in different namespace. By defining resource allocation policies, fairness can be guaranteed.
- **UnionFS** is a file system that allows files and directories in multiple file systems to form a single coherent file system. This allows efficient storage layer virtualization and fast execution environment deployment.

[1] https://www.vmware.com/products/vsphere.html
[2] https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

The light-weight virtualization technology is the enabler for serverless computing because short-lived serverless functions are less tolerant to high startup overhead.

2.2 Existing Disaggregation Prototypes

Existing resource disaggregation prototypes include HP The Machine [126], Huawei NUWA [96], Facebook Disaggregated Rack [50], Intel Rack Scale Architecture [72]. These designs often assume rack-scale resource disaggregation where resource blades only communicate with other resource blades in the same rack. Our evaluation in Chapter 4 is more general where both rack-scale and datacenter-scale disaggregation are evaluated.

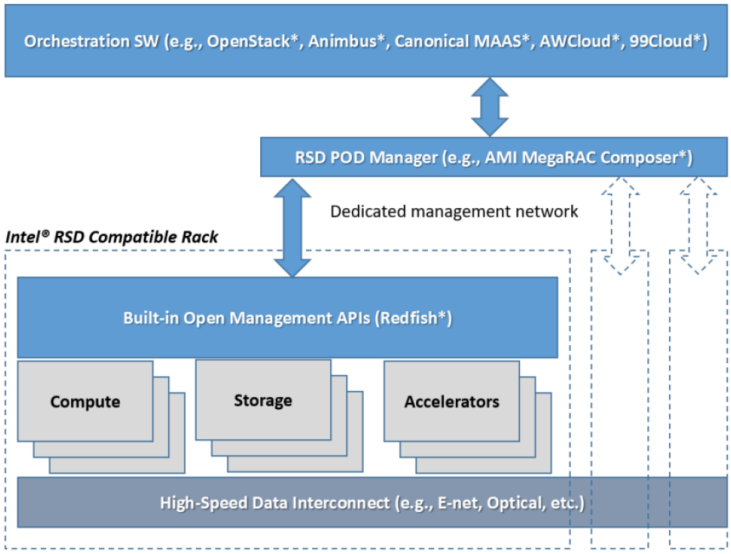


Figure 2.1: Intel RSA Design

Next, we discuss the design of Intel RSA (Rack Scale Architecture) and use it as an example to motivate our research. Figure 2.1 shows the architecture of Intel RSA. A rack is composed of resource blades rather than individual servers, and the resource blades are directly connected by the high speed data interconnect. A POD (a collection of racks) manager is connected to the management network to schedule the resources inside a POD. RSA exposes the northbound API of the POD manager to popular cluster orchestration software such as OpenStack.

Current RSA's network fabric is based on the PCIe direct attach technology. We believe this is critical to RSA's performance. However, our evaluation in Chapter 4 shows that current commodity network components are sufficient to meet the bandwidth and latency requirements of resource disaggregation.

2.3 pFabric

pFabric [16] is a flow scheduling mechanism that achieves close to optimal performance. Although pFabric requires specialized switch hardware that implements priority queues, its design is extremely simple. In pFabric, each endhost puts a number in the packet header indicating its priority. For example, the remaining flow size is used in their evaluation. Note that in this case smaller priority number has higher priority. The switch queue always dequeues the packet with the smallest priority number. In the case that the queue is full, the packet with the largest priority number in the queue is dropped unless the newly arrived packet has the largest priority number. In pFabric, flows always send at line rate and rely on packet drop to schedule flows.

Previous research [31] has established that SRPT (shortest remaining processing time) scheduling achieves lowest average job completion time. Therefore, pFabric can achieve close-to-optimal performance. However, pFabric requires specialized switch hardware that implements the priority queues, which makes the design impractical. In Chapter 5, we identified that it is not necessary to have priority queues inside the switch since most of the datacenters have full bisection bandwidth or they are not severely congested at the core. Therefore, using endhost-based schedule should be sufficient to achieve comparable performance as pFabric.

Chapter 3

An Architectural Extension for Serverless Computing

3.1 Introduction

While virtual machines have been the mainstay for cloud computing over the past decade, recent advances in lightweight isolation using containers have led to the emergence of serverless computation as a new paradigm [22, 24, 58, 100]. Serverless computation allows customers to write applications composed of light-weight, short-lived functions triggered by events. The cloud provider is responsible for allocating resources and launching these containers when appropriate, freeing the customer from the task of provisioning and maintaining long-lived virtual machines. At present, all major cloud providers (including Amazon AWS, Microsoft Azure, Google GCE, and IBM) support serverless computation, and open source efforts such as OpenLambda [64], and OpenWhisk [100] allow private clusters to implement serverless computation.

Serverless architectures benefit both cloud providers and customers. For providers, serverless computing enables fine-grained resource allocation and scheduling, which in turn leads to improved resource utilization and lower costs. For customers, serverless computing provides seamless elastic scaling. As a result, cloud providers have promoted the use of serverless computing by implementing fine-grained pricing (*e.g.*, AWS Lambda customers are charged at the granularity of 100ms units) and this architecture has been increasingly adopted by a variety of companies (*e.g.*, Coca Cola, Expedia, Samsara, EA, and others [116]).

The serverless architecture is particularly well-suited for event-based applications where event handlers are short-lived and stateless. As a result, the initial usage of serverless computing was focused on applications like web serving [116, 118], image and video compression [52, 66], and triggering builds [45]. However, due to its ease of deployment and its seamless elasticity, the serverless paradigm has been increasingly used outside of this event-based arena. For instance, serverless computing has recently been used for data analytics [20], machine learning [75], and sensor data processing [21, 117].

While the current serverless computing offerings are near-ideal for event-based applications with stateless event handlers, they do not provide support for more general forms of distributed computing. In particular, this broader set of uses would benefit from mechanisms that:

- Provide faster file I/O, because many of these new uses involve frequent file accesses.
- Handle failures in a more systematic manner, because these new uses often involve large-scale computations where failures are commonplace.
- Ensure correctness during concurrent file accesses, because these new uses often have multiple writers and readers of data.

Of course, we have long known how to address each of these issues independently; the literature is replete with standalone and general-purpose solutions for locking [40, 68], for augmenting storage with in-memory caches [80, 101], and for adding fault-tolerance to applications [120, 142]. In contrast, here we present an alternative approach called Savanna that merely inserts a single shim layer between the (serverless) applications and the storage system that provides all three of these functions in a *unified* and *simplified* manner.

Savanna is unified because, rather than providing separate interfaces for each functionality, it merely exposes a slightly modified file system abstraction. This is natural because Savanna lives between applications and storage, and so looks like a file system to the application and looks like an application to storage. The modifications are necessary to enforce consistency and failure recovery.

Savanna is simplified because, rather than supplying a fully general version of each functionality, Savanna is tailored to serverless applications (which are stateless, so can be aborted and restarted at will). This allows us to enforce a particular consistency model (snapshot isolation) with a simple locking mechanism.

By placing this functionality in between compute and storage, we have made Savanna independent of the processing framework used by an application. Thus, while in some computing frameworks (e.g., MapReduce and Spark) the failure recovery and consistency mechanisms are tied to a particular programming model, our approach here has no such limitation, enabling Savanna to be used by a wide range of applications. Also, Savanna only requires modification on the cloud service provider side.

We evaluate the generality of our approach by porting a variety of existing serverless applications and the PyWren [75] framework to use Savanna. We found that using Savanna improves application performance by $1.4\times - 6.4\times$ across these use cases. Finally, Savanna is publicly available on Github[1] and is currently in use by machine learning researchers at our institution.

[1] `https://git.io/savanna-serverless`

3.2 Background

We first give a brief overview of serverless applications, and then discuss what more recent uses of serverless computing would want from a serverless framework.

3.2.1 Serverless Applications

A serverless application consists of several *named functions* and a set of *triggers*. Each named function is implemented in a language supported by the cloud provider's serverless runtime[2] and is registered with the cloud provider (thereby associating it with a name). A *trigger* is a tuple specifying a function name and an event that should trigger the execution of the named function. Cloud providers allow triggers to specify a variety of events, including file creation, file modification, REST API calls, and HTTP accesses.

In the rest of this chapter we refer to a particular execution of a named function (triggered by an event) as an *invocation*, and use the term *job* to refer to a set of invocations associated with the same overall goal. An invocation is dispatched by a coordinator and is executed on any available server. Each invocation can run for a bounded period of time (between 5–10 minutes depending on provider) after which it is killed.

Invocations are executed within a container [47,86] which is responsible for providing isolation and ensuring that different invocations of the same function run within a consistent environment. Cloud providers do not allow developers to specify placement constraints or preferences. However, serverless infrastructures rely on container reuse [133] to reduce latency when handling an event; as a result invocations from the same user are commonly placed on the same server. Therefore, container reuse help us improve the efficiency of our caching strategy.

Since invocations are short-lived, named functions are stateless and must rely on external storage services for any long-lived state. In current serverless architectures, named functions commonly make use of blob stores such as Amazon's S3 [112], Azure's Blob Store [25], and Google Cloud Store [57], and rely on third party libraries (*e.g.*, Boto [39]) to save and retrieve state.

3.2.2 New Serverless Workloads

Serverless computing frees users from having to allocate virtual machines or containers, and from configuring the software stack. As a result several academic and industrial efforts are looking at moving new workloads – such as data analytics and machine learning – to serverless environments. These efforts include: PyWren, a framework for scientific computing users to use the cloud; Databricks Serverless [2], a commercial effort to enable data analytics in serverless environments; efforts by Google and Amazon enabling serverless deployment of TensorFlow [3, 7], a popular machine learning framework; and Amazon's AWS GreenGrass, a service that enables IoT devices

[2]For example AWS Lambda supports functions written in Python, Node.js, C# and Java. Similarly, Google's Cloud Functions supports Node.js; and Azure supports C#, F#, PHP, Python, Bash and PowerShell.

to invoke Lambda functions for processing sensor data. These emerging applications have placed new requirements on serverless computing.

3.2.3 Requirements from New Workloads

Serverless computing is commonly used for short-lived event responses, which typically have low concurrency and I/O intensity. For these uses, the current serverless storage solutions (using blob stores) are sufficient. However, many of the more general serverless workloads involve higher concurrency and require better I/O performance, and current cloud blob stores fall short of providing the desired consistency, fault tolerance, and high throughput (as noted in [75]). We discuss each of these issues below.

Consistency. Cloud blob stores, such as Amazon S3 and Google Cloud Storage, provide a read-after-create consistency model[3]. This only guarantees that a newly created file is visible immediately by others; updates to a file are only eventually consistent (i.e., there are no guarantees about when updates are visible to readers, nor that they become visible to different readers at the same time). This model has been sufficient for short-lived applications with limited concurrency. However, for more heavyweight applications with a higher degree of concurrency, this loose consistency model can lead to incorrect executions.

A layer between compute and storage is the logical place to enforce the locking primitives needed to enforce various forms of consistency. As we describe in §3.3.5, Savanna utilizes locking to enforce snapshot isolation.

Fault-Tolerance. Failures are common in the cloud environment, especially for large parallel jobs using thousands of cores. While current serverless compute framework, such as Amazon Lambda, retries the invocation on a failure [10], the current serverless storage model does nothing to help applications recover data from such failures. For instance, if a named function begins its execution, stores some intermediate results in a shared file, and then fails before completion, the blob store will be left in an indeterminate state with no way to reconstruct the version of the shared file as it existed before the invocation of that named function. Thus, one would have to restart the entire job from scratch. Obviously, upon a failure, one would prefer to only restart the failed invocations instead of the entire job. This would require invocations to be atomic; either they run completely and their results are visible to all, or they fail and none of their results are visible.

Since this atomicity is not supported by current serverless offerings, developers would need to implement the logic (including logging and undo functionality) required to rollback the effects of any failed invocations before restarting them, which increases application complexity and decreases performance.

In serverless architectures, a layer between compute and storage can control which reads and writes are allowed, and when writes are made visible to others in the persistent storage. Further,

[3] Azure Storage has a stronger notion of consistency, but it is per-file. As we discuss later in §3.3.3, our notion of consistency – snapshot isolation – applies across files

cloud providers have fine-grained control over the scheduling and invocation of a named function, so that invocations can be restarted when necessary. Thus, as we show in §3.3.5, this allows Savanna to provide fault tolerance to applications.

I/O Throughput. Compared to a local file system or a cluster file system (*e.g.*, HDFS, Ceph), where compute and storage can reside on the same server, a cloud blob store is of larger scale and is typically disaggregated from the compute resources. Since a named function is stateless, all of its outputs must be persisted to the cloud blob store after it finishes. However, the access to the blob storage has limited throughput. For instance, a recent measurement [75] showed that Amazon limits the throughput of a connection to S3 to around 30MB/s, and Azure Blob Store has a 60MB/s limitation [26]. Further, Amazon imposes a limitation of 100 PUT and 300 GET requests per second per customer [113], which reduces shuffle performance of MapReduce jobs because shuffle between m mappers and n reducers can generate $m * n$ shuffle files.

The decoupling of compute and storage increases the ease of programming, but it comes at performance cost. In order to overcome both the natural limitations of accessing remote storage (losing locality), and the imposed limitations (as described above), Savanna includes a simple caching layer that resides on the server. This makes I/O faster, and reduces the load on the cloud blob store, meeting the needs of both users and cloud providers.

3.3 Savanna Design

Savanna provides a simple and unified file API (§3.3.1) to the programmers, and encapsulates the complexity of consistency, fault-tolerance and caching in the interface. Savanna relies on a traditional blob store for durable storage and we assume that values written to a blob store by Savanna are not modified by any external application.

To achieve the various fault-tolerance and consistency properties, which we describe in more detail below, we impose several constraints on programs. However, in each case these are either straightforward, or similar to the constraints imposed by stream processing frameworks such as Spark [143], Flink [51], and Naiad [92]. These constraints are:

- We assume that named functions are deterministic (when rerun with the same inputs, they produce the same outputs).
- We assume that if a named function accesses data outside of Savanna, such data is either immutable or does not affect its outputs.
- We assume that Savanna can detect an invocation's successful completion (referred to as an invocation commit, which naturally happens when the named function returns), and an invocation's failure (referred to as an abort, which happens if the named function throws an exception or fails to return before some preconfigured timeout value; in the latter case, Savanna explicitly kills the named function).

- We assume that inputs to a named function can be recreated; thus, input files must not be deleted until changes have been committed to the backend blob store, or services like Kafka [76] must be used to log stream events.

Moreover, similar to existing distributed frameworks, we assume fail-stop behavior for server errors, and do not handle byzantine behavior due to partial failures of servers. This can be enforced using either the `watchdog` daemon in Linux or through the use of failure detectors such as Falcon [79].

3.3.1 Programming Model

Each invocation in Savanna is assigned a *logical start time* that is globally unique and monotonically increasing (i.e., an invocation that arrives earlier by wall clock time has a lower start time).

Savanna appears as a new file API to serverless developers, with functions that can be used to open or close files, create a new file, read from a file and write to a file.[4] Syntactically, these functions look identical to existing file system and storage APIs, but have different semantics as listed below:

- **Open:** An invocation can successfully open a file if the file has not been written to by an invocation with a later logical start time. An invocation aborts when a file cannot be opened due to this requirement, and is restarted by the fault handling mechanism (§3.3.5).
- **Read:** We do not modify the semantics of read, but prefetch data to improve performance.
- **Write:** All writes from an invocation are buffered, and become visible if and only if the invocation commits. To avoid aborting invocations that have successfully completed, we use a per-file reader-writer lock to ensure that any invocation that opens a file for writing has exclusive access to the file. Finally we also guarantee atomicity for writes ensuring that either all of an invocation's writes are visible (to other invocations) or none of them are visible.

3.3.2 Savanna Components

The semantics described above are implemented using a centralized metadata coordinator and a set of distributed Savanna agents (Figure 3.1).

Metadata Coordinator. The centralized metadata coordinator is shared by users and jobs, and is responsible for tracking invocation metadata, such as which named function is being run in an invocation, what server the invocation is running on, and its inputs, wall clock start time, and logical

[4] We inherit our access control policies from the cloud provider's blob store, but in what follows we assume that invocations have permission to access the relevant files.

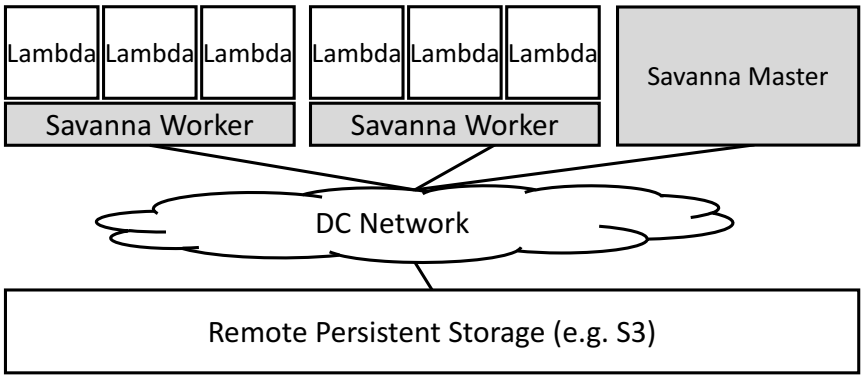


Figure 3.1: Savanna consists of a logically centralized metadata coordinator and a set of distributed Savanna agents, and relies on the cloud provider's blob store for durable storage.

start time. The metadata coordinator also implements an atomic counter that is used to allocate logical start times for a new invocation; in our current implementation this is a simple counter and the logical start time corresponds to the order in which invocations arrive at the metadata coordinator.

The metadata coordinator is also responsible for tracking file metadata, in particular (a) the location of a file, which can either be a server (where data is cached) or a blob identifier in a blob storage service like S3, (b) the logical start time of the last modifying invocation, (c) the files which were accessed by the last invocation that wrote to that file (which we refer to loosely as *lineage*), and (d) the per-file reader-writer lock used to ensure exclusive access for writers. Since an invocation can only run for a bounded length of time, all locks in the metadata coordinator act as leases [59] (i.e., they are released after a timeout period). This ensures that file locks are eventually released despite server failure or network partitions. Finally, we note that the metadata coordinator can be distributed (for fault tolerance or scaling) using standard replicated state machine (RSM) based techniques; however our implementation currently uses a centralized version. While our system can be extended to survive metadata coordinator failures by reconstructing state from the blob store and Savanna agents, for simplicity our current design assumes that we do not lose content stored in the metadata coordinator.

Savanna Agent. We run a Savanna agent on each server in the datacenter; these Savanna agents are responsible for coordinating with the metadata coordinator for file accesses, detecting when an invocation commits or aborts, and implementing Savanna's caching layer as described in §3.3.4. When a named function is invoked, the Savanna agent communicates with the metadata coordinator to record metadata for this invocation and to retrieve the current logical time. Invocations communicate with the Savanna agent when performing file operations. The Savanna agent communicates with the centralized metadata coordinator whenever an invocation attempts to open a file, and aborts the invocation if the file cannot be opened without violating the semantics specified in §3.3.1.

All file writes are buffered by the Savanna agent in an invocation-specific buffer that cannot

be accessed by any other invocation in the system. When an invocation signals that it has finished running and can commit, then the Savanna agent persists changes in this buffer and appropriately updates file metadata. Whenever an invocation signals completion (i.e., signals that it can commit or must abort) the Savanna agent uses standard mechanisms to kill the invocation and release all file locks held by the invocation. Finally the Savanna agent is also responsible for implementing Savanna's *cache* (§3.3.4).

The metadata coordinator and Savanna agents described above implement the semantics described in §3.3.1, and as we show next these semantics are sufficient to provide snapshot isolation and fault tolerance.

3.3.3 Consistency

Named Functions in Savanna are assumed to be written in an imperative language (*e.g.*, Python), and we assume that the set of files accessed by a named function cannot be determined statically. Given this limitation we cannot provide strict serializability without sacrificing all concurrency in the system. As a result, we instead provide snapshot isolation, a weaker consistency guarantee which ensures that all writes made by an invocation are atomically visible (i.e., another invocation either sees all of the writes made by the invocation, or none of them), and that the invocation only reads data that was written before it started. A recent survey [28, 29] found that snapshot isolation is the strongest form of isolation implemented by several commercial databases (including Oracle and SAP Hana). Therefore, Savanna's consistency guarantees should be sufficient for most serverless applications. As we show in §3.5, for many applications our implementation imposes minimal overhead when compared to the current serverless architecture. However, snapshot isolation might affect the performance of applications with significant write contention. But any application with contended writes needs to coordinate file access to avoid overwriting changes, and the same coordination mechanisms can be used to prevent aborts in Savanna, thereby improving performance.

Snapshot isolation ensures two properties:

- **Snapshot reads:** an invocation cannot read data written by a future invocation, and
- **Atomicity:** writes from an incomplete or aborted invocation are never read by another invocation.

Atomicity is trivially satisfied by Savanna's semantics (§3.3.1). To see that Savanna also provides snapshot reads, consider two committing invocations a_0 and a_1 with logical start times t_0 and t_1. Further consider the case where $t_0 < t_1$ and a_0 reads from some file f, while a_1 writes to the same file f. We claim that a_0 cannot read changes to f made by a_1. This is of course trivially true in any case where a_0 commits before a_1 starts. In the case where a_0 and a_1 run concurrently the reader-writer lock serializes access to the file and one of a_0 or a_1 needs to commit before the other can proceed. Consistency is trivially satisfied if a_0 commits before a_1. However, if a_1 commits first then a_0 must abort due to the semantics of open as defined in §3.3.1. Finally, if a_1 commits

before a_0 starts then the open semantics ensure that a_0 will abort when it attempts to open f. Thus we can see that Savanna ensures that any committed invocation with logical start time t_0 must have only read data written by an invocation whose start time was less than t_0.

3.3.4 Caching

Savanna relies on caching to reduce the latency of accessing a blob store. We decided to place a shared-cache on each named function server rather than a local cache inside each container or a lookaside blob cache for the following reasons. Placing the cache on the hosting server rather than inside the containers maximizes sharing between containers and different containers can share the cache with zero overhead. Comparing with a lookaside blob cache, Savanna's design enables colocation of compute and data, where named functions can be placed on the server with its input data.

Savanna lazily synchronizes cached objects to the backend blob store by the Savanna agent. Invocations always write data to the local cache and attempt to read data from the local cache. If the file does not exist in the local cache, the Savanna agent contacts the metadata coordinator to obtain the address of a neighbor Savanna agent who has the most updated version of the file. The local Savanna agent connects to the neighbor Savanna agent to obtain a copy of the file. In cases where the most recent copy of the file is available in the blob store, a local Savanna agent can choose to fetch the file by either accessing a cached copy on another Savanna agent or by directly accessing the blob store. Finally, Savanna agents also implement prefetching, which we found to be useful for applications that rely on sequential data access.

3.3.5 Fault Tolerance

Our fault tolerance mechanisms are implemented by both the Savanna agent and the metadata coordinator, and are designed to ensure that (a) invocations are retried until an event is successfully processed, (b) file locks are eventually released, and (c) data is not lost due to server failures or other events that result in parts of the cache being lost. Note that we assume the backend blob store is durable and as a result our fault tolerance mechanisms do not handle failures in the blob store.

In Savanna, invocations might fail or be aborted before they have committed, but we now argue that they will have no externally visible effect on Savanna's persistent storage system. When an invocation aborts, the Savanna agent contacts the metadata coordinator to record a new invocation of the same named function with the same inputs. Note this new invocation has a different *start time* than the aborted invocation, and hence operates on a different snapshot of the storage system. The Savanna agent then retries this invocation on the same server.[5] An invocation may also fail to be processed due to failures in the server where the event was being processed. We handle this by having the metadata coordinator trigger a recover invocation for an invocation whose metadata has not been updated by an Savanna agent within a configurable period of time. Since invocations

[5]As is standard, we report an error and halt processing if an event requires more than a configured number of invocations to be processed.

Event	Cause	Action
Abort	Violating snapshot read	Restart on the same machine with new *start time*
Fail	Server failure	Recover the invocation and any failed ancestor invocations with their old *start time*

Table 3.1: Two types of named function restart

can run for bounded time this retry clock can be aggressively set to be only slightly larger than the invocation runtime, thus improving the system's responsiveness to server failures. Table 3.1 summarizes the behavior under abort and fail.

Finally, we rely on lineage-based reconstruction to handle data loss due to server failures. This avoids expensive replication-based data recovery [101] and reduces cluster memory usage. Savanna stores multiple versions of a file, and uses previous versions to recover from data loss. When writing a modified version of the file, the Savanna agent includes a list of all files accessed by the invocation and their last modified times. Upon observing data loss, Savanna uses this information to recreate the file as follows: first, it looks up the file's last modified time to find the appropriate invocation metadata which records information about the event that triggered the change, and the named function that was activated; next, it launches a recovery task on a Savanna agent by providing it with (a) the invocation metadata and (b) the file's lineage; finally the Savanna agent executes the invocation using the versions of files recorded in the lineage rather than the current (potentially newer) versions. Furthermore, reads and writes for a recovery invocation are guaranteed to not abort since it is emulating the behavior of a previously committed invocation. Since we assume that invocations are idempotent and deterministic this recreates lost data.

3.4 Implementation

Savanna's core code is implemented using C++, and a Python client is provided to the programmer to use the Savanna API. Savanna is a generic extension designed to work with any serverless framework. To demonstrate its portability, we have integrated Savanna into two popular open source serverless frameworks – OpenWhisk [100] and PyWren [75] – with little effort. We have made Savanna source code available on Github for future research and development. For a better understanding of the implementation details, we discuss some critical optimizations that improves Savanna's performance in this section.

Savanna Metadata Coordinator. The metadata coordinator must be able to handle thousands of concurrent connections in parallel. We use `epoll` along with a thread pool to manage the connections, and the *power of two choices* [90] rule is used to assign new connections to threads. To maintain a dictionary of file metadata with frequent insertion, the concurrent hash map in Intel Thread Building Blocks [73] (TBB) is used. This is found to be useful for write heavy workloads.

Savanna Agent. The Savanna agent is in charge of fetching data from S3 if a file is not cached.

A process pool is used to fetch the data and feeds the data to the invocation that consumes the data. This enables better compute and I/O pipelining, which improves application performance in practice. We limit the size of the process pool to 3 per invocation such that we do not exceeds the cloud blob store's (*e.g.*, S3) request rate limit.

Cache Storage. Savanna can be configured to store cached files in shared memory or a local disk-based file system. Using a disk-based file system does not significantly degrade Savanna's performance due to modern file systems' write-back behavior – a file is cached in memory before it is written to disk storage. We evaluate this by running workloads on `i3.8xlarge` with fast local SSDs, and find that a disk-based file system only slows the workloads by 6% compared to the shared memory storage. In the rest of the paper, we evaluate Savanna performance using the shared memory storage (`/dev/shm`).

Garbage Collection. We ensure that Savanna's storage requirements do not grow without bounds through a simple garbage collection strategy. We clear the lineage of any file that has been synchronized to the persistent store. Savanna also periodically scans old versions of a file which are not referenced by any lineage and deletes them, thus reclaiming space. In the rare case that a single invocation suddenly fills up the local cache storage, Savanna pauses the `write` function call, and starts the garbage collection processes to obtain more cache space.

Cache-only Files. Based on user feedback, it is useful to have cache-only temporary files. For example, the shuffle files of a MapReduce job will be only read once, hence do not need to be persisted to the cloud blob store. When programmers open files to write, they can indicate the file is a cache-only file (with the number of reads before expire or an expire time). These files will not be persisted to the blob store, with an exception when garbage collector evict them to the blob storage on severe memory shortage.

Deployment. Savanna requires a modification on existing platform where each worker machine needs to launch the Savanna agent and a metadata coordinator is required in a cluster. We have integrated Savanna with PyWren and OpenWhisk with very little effort. Each named function needs to import the Python client in order to use Savanna API. As we can see, only the cloud service providers need to modify their serverless execution platform. From cloud customers' perspective, Savanna is a transparent layer.

3.5 Evaluation

We start our evaluation in §3.5.1 by looking at the performance of both traditional serverless applications such as file converters and a set of newer serverless applications such as Data Analytics, Machine Learning, Internet of Things (IoT), and Data Streaming. In the following subsections we focus on specific properties of Savanna: consistency (§3.5.2), fault tolerance (§3.5.3), scalability (§3.5.4), and its behavior under pathological conditions (§3.5.5). We end this section with a case study (§3.5.6) in which Savanna was used by machine learning researchers from our institution to

Name	Abbr.	Description	Parallelism	Category
Big Data Benchmark Query 1	BDB_1	Scan and filter query on 5.2GB page URLs [33]	100	Data Analytics
Big Data Benchmark Query 2	BDB_2	Aggregation query on 123GB data [33]	160	Data Analytics
Big Data Benchmark Query 3	BDB_3	Join Query on 123GB data [33]	160	Data Analytics
Sort Benchmark	Sort	Sort 160GB 100Byte Records [122]	160	Data Analytics
Large Scale Matrix Multiplication	Matrix	Generate 250GB kernel matrix from 64GB feature matrix [129]	900	Machine Learning
Parameter Server	PS	A Parameter Serve hosting 300MB model [82]	160	Machine Learning
Image Object Detection	OD	Detecting objects in 30 images using Darknet [109]	1	IoT
Time Series Cosine Similarity	TS	Time series cosine similarity using smart electricity meter data [121]	160	IoT
Popular Twitter Hashtag Stream	Stream	Find popular recent Twitter hashtags from a stream [130]	1	Traditional
Image Compression	IMG	Compress newly uploaded image [66]	1	Traditional

Table 3.2: Workloads used in evaluation. Parallelism refers to the concurrent invocations per workload and our cluster has 16 slots per server.

accelerate their computation.

3.5.1 Performance

We evaluate Savanna by running 10 workloads on Amazon EC2, and compare Savanna's performance against using S3 directly. With Savanna, application performance improves between 1.4× and 6.4×.

Experiment Setup. We evaluate Savanna on Amazon EC2. Since Savanna Agents must be deployed on each worker node, we set up our own serverless service, which allows us to host named functions in a controlled EC2 environment. To do a fair comparison, we compare Savanna with the baseline of our custom serverless service that uses S3 as its blob store. The service is installed in a cluster with 10Gbps connections (`r4.8xlarge`), with Virtual Private Cloud (VPC), Enhanced Networking and S3 VPC Endpoint enabled. Each cluster contains a metadata coordinator node and a bunch of worker nodes that run Savanna Agents and invocations. Each invocation is allocated with two EC2 Computing Units, which is roughly equal to two hyper-threads or one physical core. This resource allocation scheme is consistent with Amazon Lambda's allocation [22]. Finally, all of our experiments run 16 invocations per machine.

Workloads. We study 10 workloads from various categories, implementing these applications with Python 2.7, and using Savanna's Python client or Boto3 for file I/O. The applications are summarized in Table 3.2.

First, we evaluate Savanna using the Big Data Benchmark [33], which is commonly used in recent papers [18,54,102] to study data analytics systems. We evaluate three queries (query 1c, 2c, 3c, the most intensive workloads in their types) with common SQL operations such as filtering, aggregation and join, and use two input datasets - `Rankings` (5.2GB) and `UserVisits` (117GB). We also implemented the sort benchmark [122] which sorts 160GB of 100 byte records using the TeraSort [103] algorithm.

For machine learning workloads, we evaluate Savanna using a kernel matrix multiplication workload [129]. This workload consumes feature vectors and uses matrix multiplication to gen-

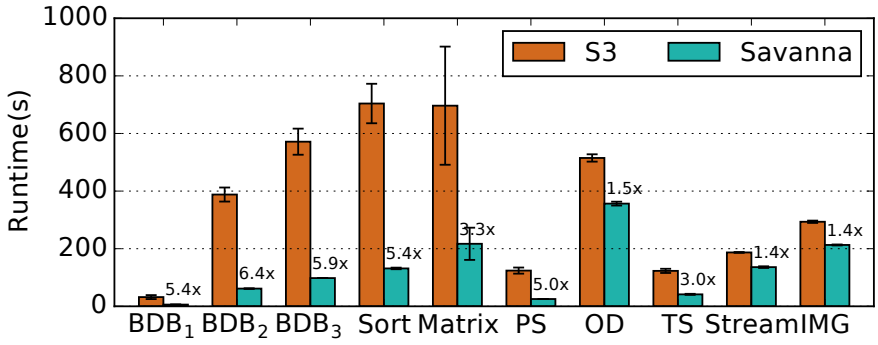


Figure 3.2: Savanna improves data analytics workloads and machine learning workloads by 3.3× to 6.4×. For IoT and traditional serverless workloads, it accelerates their speed between 1.4× and 3.0×. Savanna also improves the stability of the system as the runtime varies less across runs (smaller error bars).

erate a kernel space representation of a machine learning model. The workload was originally implemented using PyWren, which is what motivated our Savanna PyWren port. More details on this can found in the case study in §3.5.6. We also implemented a parameter server [82] workload for distributed machine learning. We emulate the process of gradient updates from distributed GPUs to a 300MB model. The consistency property of Savanna guarantees each gradient update is atomic and is never overwritten by other peer GPUs.

We evaluated Savanna for two IoT-style applications: an image object detection workload and a time series analysis workload to show that even with less data intensive workloads, Savanna improves performance. Lastly, we evaluated two traditional workloads for serverless computing – an event handler that analyzes popular Twitter hashtags and a file converter that compresses user-generated images.

By varying the degree of parallelism of some workloads, we verified that the high parallelism in our workloads is not causing any I/O bottlenecks in the baseline. The lower performance number in our baseline is mainly because of the limited throughput of S3.

Performance Improvement. We now summarize Savanna's performance improvements relative to S3. We also attempted to evaluate AWS Elastic File System (EFS), but we find its throughput fails to scale for most of the workloads [1]. For all the workloads, the cache is always cleared before each run so that all input data is read from S3 rather than Savanna cache.

Data analytics workloads achieve 5.4× to 6.4× performance improvement. The improvements are mostly coming from prefetching, asynchrounous writes, and the cache-only temporary files - in this case the shuffle files generated by mappers and consumed by reducers. By avoiding writing to S3, which only has 30MB/s average throughput per thread, Savanna improves these workloads' performance.

For machine learning workloads, Savanna improves the performance from 3.3× to 5.0×. In

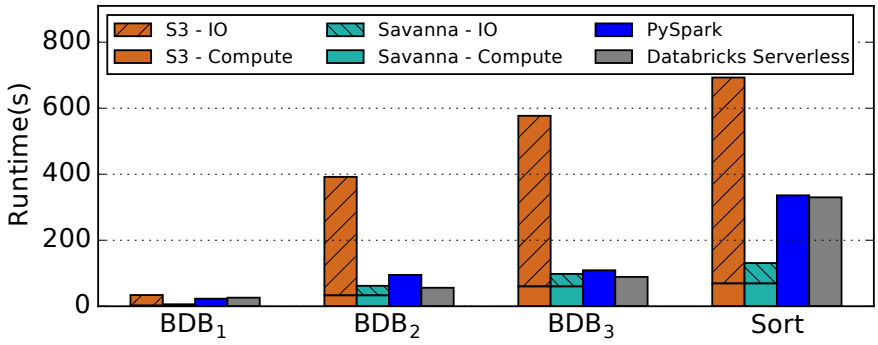


Figure 3.3: Savanna reduces I/O time from 89%-94% to 38%-66% and it significantly outperforms PySpark and Databricks Serverless under the BDB_1 and Sort workloads.

these workloads, the same data object is often consumed by multiple subsequent invocations which makes caching effective. For example, in the Large Scale Matrix Multiplication workload, the input matrix is partitioned into many chunks. Each chunk must be multiplied with other chunks in parallel and caching the chunks significantly improves I/O time. In Savanna, multiple invocations attempt to lock the chunk, and the winner fetches the chunk from S3 and cache the data in its Savanna agent, where the other invocations could read from. Although in machine learning workloads, cached data can be efficiently reused, we observe lower speedup because of the compute-intensive nature of these workloads (hence less time is spent on I/O).

We find there is less performance benefit for IoT and traditional serverless workloads. These workloads are usually very compute intensive, hence improving I/O does not reduce the overall runtime by much. However, Savanna still improves their performance by 1.4× to 3.0×. While some of the performance benefits are attributed to caching (*e.g.*, avoid fetching a 200MB neural network model in the object detection workload), these workloads achieve better performance mainly by prefetching and lazily writing back to S3. Recall that Savanna uses a background process to prefetch data from S3 and to asynchronously write data to S3. Prefetching improves compute and I/O pipelining and writing back allows returning the invocation when outputs are cached, rather than waiting for them being persisted to S3. If the server fails before the outputs are persisted, lineage is used to reconstruct the lost data.

Data Analytics Workloads Performance. We now look more closely at the data analytics workloads. We instrumented these workloads to measure the compute time, and attribute the remaining runtime of each workload as I/O time (which we know is only approximately correct, but this still serves as a useful performance diagnostic). Figure 3.3 plots the performance of the data analytics workloads using Savanna and S3, with their runtime decomposed into compute time and I/O time. Using S3 results in spending 89% to 94% of the time waiting for I/O due to low S3 performance. With Savanna, the I/O time drops to 38% to 66% of the overall runtime.

Next we compare Savanna's performance against Apache Spark and Databricks Serverless, the state-of-the-art data analytics engines. Currently Savanna is restricted to tasks written in Python,

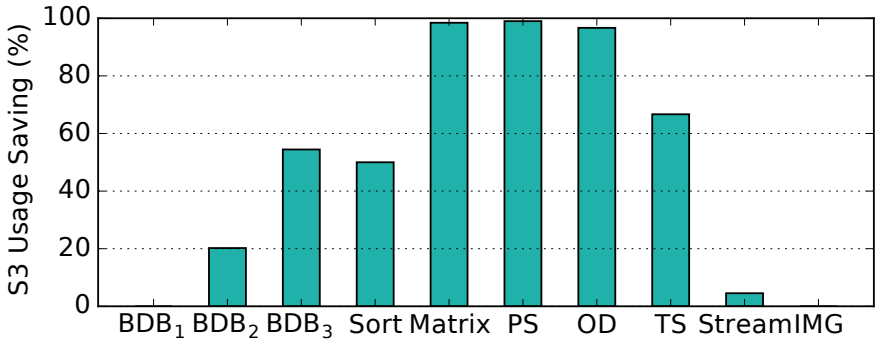


Figure 3.4: Savanna reduces S3 usage. Therefore, both cloud customers and providers have motivation to deploy Savanna.

and hence we compare our performance against that of a similar job written using PySpark. Furthermore, we use S3 files, read using the S3A interface [114], as input for the PySpark job, and write outputs to local disk. For Databricks Serverless, the input and output files are both read from and written to S3.

In Figure 3.3, we find that Savanna outperforms PySpark in all evaluated workloads, and it is faster than Databricks Serverless under the BDB_1 and Sort workloads. Note that Databricks Serverless only helps the users manage their VMs with a restricted programming interface. By contrast, Savanna is a more general framework that is capable of processing more than BSP (Bulk Synchronous Parallel) workloads, with extra features of providing consistency and fault recovery.

S3 Usage Reduction. By caching temporary files (indicated as cache-only by the programmer) and frequently accessed files, Savanna not only improves application performance, but also reduces S3 usage. Therefore, both cloud customers and providers have an incentive to deploy Savanna, as most major cloud providers have limitations on blob store request rates.

Figure 3.4 plots the S3 usage reduction after using Savanna. We find that all data analytics and machine learning workloads have a large usage reduction, with one exception – the BDB_1 workload. Since most of these workloads are I/O intensive, it is natural that fewer accesses to S3 improves I/O performance. BDB_1 has only one stage (thus no reads from cache), so there is no bandwidth saving, and the performance benefit is due to prefetching and asynchronous writes. For IoT applications, we also observe high S3 bandwidth savings. However, these workloads have lower performance gain as they are all very compute intensive. Traditional serverless applications have nearly no bandwidth saving and hence their performance gains are lower.

Savanna Raw I/O Performance. From our previous analysis, Savanna reduces S3 accesses when there is a cache hit. We now evaluate how much a cache hit is faster than a S3 access using two workloads from YCSB [43].

We evaluate YCSB A and YCSB B, two representative workloads from the benchmark with file sizes varying from 1KB to 100MB. YCSB A is a write heavy workload with 50% writes and 50%

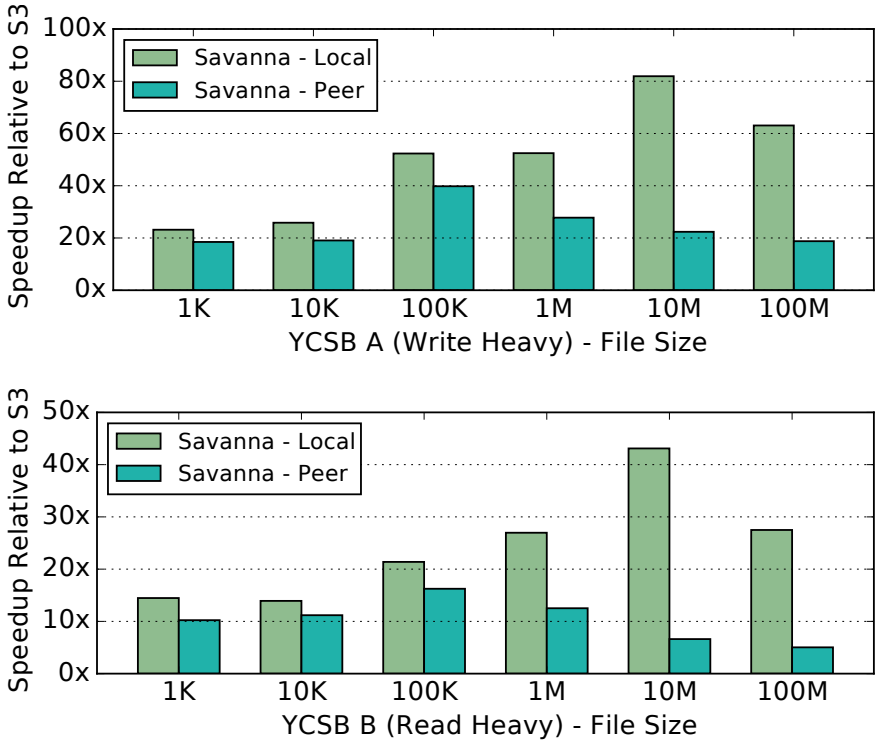


Figure 3.5: We evaluate YCSB A (50% writes and 50% reads) and YCSB B (5% writes and 95% reads) with varying file size. Savanna improves the I/O performance by 14× and 82× with local cache hits, and 5× to 40× with peer cache hits. Therefore, Savanna improves application performance on workloads with thousands of invocations (and spans multiple machines) because effective peer cache.

reads, and workload B is a read heavy workload with 5% writes and 95% reads. We evaluate both workloads and compare Savanna with S3 by plotting the relative speedup in Figure 3.5[6]. Since there are two types of cache hits (local and peer) in Savanna, we plot separate bars in the figure. A local hit is the case when the file to read happens to be on the local machine's cache. A peer hit is when the data exists in some Savanna Agent's cache, so the invocation must read from the network.

As expected, local cache hits have larger performance gains. Depending on the file size, Savanna improves I/O performance between 14× and 82× on local hits. Realizing that a local cache hit does not always happen [7], we evaluate the peer cache hit performance, where most of the time the data is cached on a neighbor Savanna Agent. Although a peer cache hit includes a network I/O, we find that Savanna is still 5× to 40× faster than S3. This explains why Savanna improves ap-

[6]Note that Figure 3.5 shows Savanna's relative speedup to S3 rather than its absolute performance, this explains the drop of speedup when file size is larger. (*e.g.*, Savanna - Local in YCSB A)

[7]A smart scheduler could be implemented to place an invocation on the machine that has a copy of its input data. However, our current scheduler only does random placement.

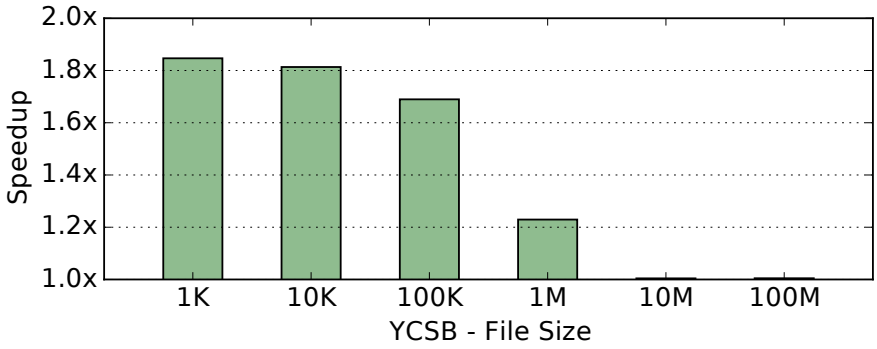


Figure 3.6: Disabling consistency only improves small files' performance by a little. For files larger than 1MB, disabling consistency introduces no benefit. In a typical cloud environment where file sizes are often larger than 1M, Savanna's consistency feature could be enabled by default.

plication performance for workloads that span multiple machines, because reading from a peering server still improves performance by 5× to 40×.

We find that both Savanna and S3 have lower throughput for smaller files. For Savanna, each I/O operation includes a lock and an unlock operation in the metadata coordinator, which adds two network round trips. This cost can be amortized for large file I/O, but not with smaller files.

3.5.2 Consistency

Savanna speeds up serverless applications by providing high throughput I/O, but it also provides consistency guarantees. We now evaluate the performance impact of consistency.

To do so, we implement a cache only version of Savanna with the consistency feature disabled. We then compare its performance against the full-fledged Savanna on the YCSB A and YCSB B workloads. Since both workloads exhibit similar performance characteristics on Savanna, for clarity and brevity we only show the result of YCSB B (the read heavy workload) in Figure 3.6. Figure 3.6 shows the speedup after disabling consistency. For small 1K files, disabling consistency reduces lock contention and leads to a 1.8× speedup. However, for files larger than 1MB, we do not observe any speedup.

The natural question to ask, then, is what is the typical file sizes in cloud environments. The answer is that cloud file systems usually handle large file chunks. For example, the default file block size of HDFS is often 64MB or 128MB. Smaller objects are usually stored in databases instead of file systems or blob stores. We therefore believe that the cost of providing consistency in Savanna is negligible for most use cases.

We validate our assumption by running real workloads. Figure 3.7 plots each workload's performance on Savanna with and without consistency. We do not observe significant performance impact of consistency in these results, which is natural because these workloads typically involve large files, with negligible consistency overhead.

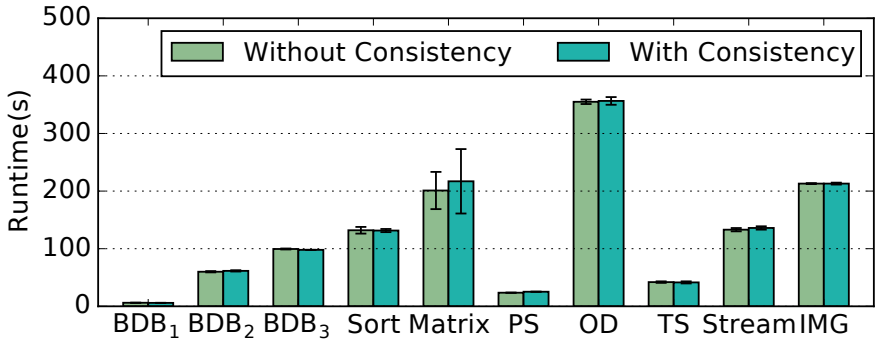


Figure 3.7: In real workloads where file sizes are often large, there is no observable performance degradation when consistency feature is enabled.

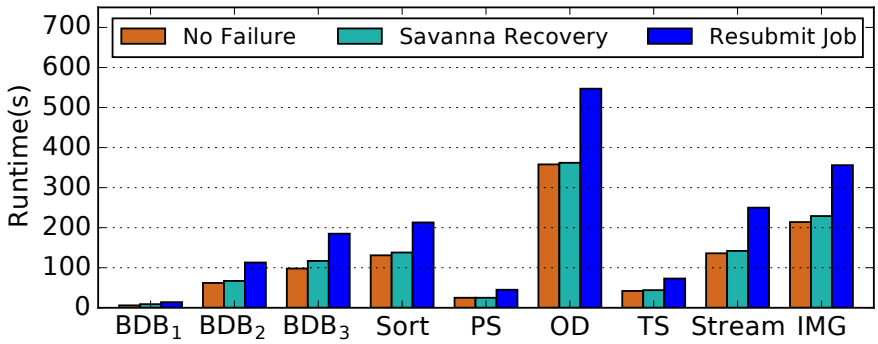


Figure 3.8: The runtime of a no-failure run, a Savanna-recovery run, and resubmitting the entire job. The failures are triggered roughly half way through the executions, so resubmitting job after failure takes about 50% more time than the no-failure case.

3.5.3 Fault Tolerance

The consistency guarantee, which ensures that invocations are atomic, allows us to treat each named function as a transform of a subset of S3 data. By logging each transformation and keeping the lineage, lost data and failed tasks can be recovered at low cost. We evaluate the performance of failure recovery with the following settings. For each workload, we kill all the named function worker processes on one randomly-picked machine half way through a job. Upon failure, Savanna detects the failed tasks of the job and restores them on another backup machine. We compare the runtime of using Savanna recovery with the "no-failure" case and with the resubmitting the entire job (i.e., all its tasks) case. Note that the matrix multiplication workload is running on PyWren, and Savanna currently does not support recovery on PyWren, so it is omitted from these results.

Figure 3.8 illustrates the runtime of all the three cases. Savanna recovery is just slightly slower than the run without failure. Resubmitting the job often incurs 50% overhead since the failures are triggered at the halfway point of the jobs. More importantly, resubmitting the job may result in incorrect outputs, as the original input may have been overwritten by the failed job. For example,

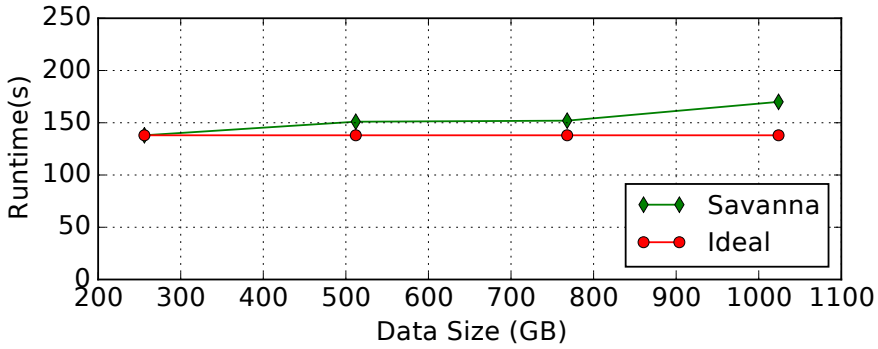


Figure 3.9: Measuring Savanna's weak scaling performance by sorting 256GB, 512GB, 768GB and 1TB of 100 Byte records. The input data size to concurrent invocation ratio is kept at 1GB/invocation.

a function that intends to increment a matrix by one may end up with two increments if the entire job is resubmitted. However, Savanna keeps track of the lineage so such errors will not occur.

3.5.4 Scalability

Next, we evaluate Savanna's scalability in terms of *weak scaling* and *strong scaling*. Weak scaling evaluates how Savanna's performance varies with input size while holding the input to compute resources ratio fixed. Strong scaling evaluates how Savanna's performance varies with compute resources while holding the total input fixed. We only evaluate the cases with no shared writes to focus on the overhead introduced by the Savanna framework. A workload with contented writes on a small set of files can scale poorly, but that is not inherent to Savanna's performance.

Weak Scaling. We evaluate Savanna's weak scaling by using input size of 256GB, 512GB, 768GB and 1TB with proportional number of invocations. For example, 256 concurrent named function invocations are used for the 256GB workload and 1024 concurrent invocations are used for the 1TB workload. Figure 3.9 shows Savanna's runtime along with the corresponding perfect scaling curve over varying input sizes. Savanna's performance is only slightly slower than the baseline when sorting 1TB data, so its weak scaling is near-ideal.

Strong Scaling. We evaluate strong scaling by sorting 512GB data, and we vary the number of concurrent invocations from 32 to 512. Figure 3.10 shows Savanna's strong scaling performance, in which Savanna's runtime curve overlaps with the ideal scaling curve most of the time.

3.5.5 Local Storage Constraints

Savanna's garbage collection mechanism removes infrequently used files from the cache, and lazily synchronizes cached files to the blob store to keep enough space for cache writes. We consider two cases where garbage collection may impact application performance.

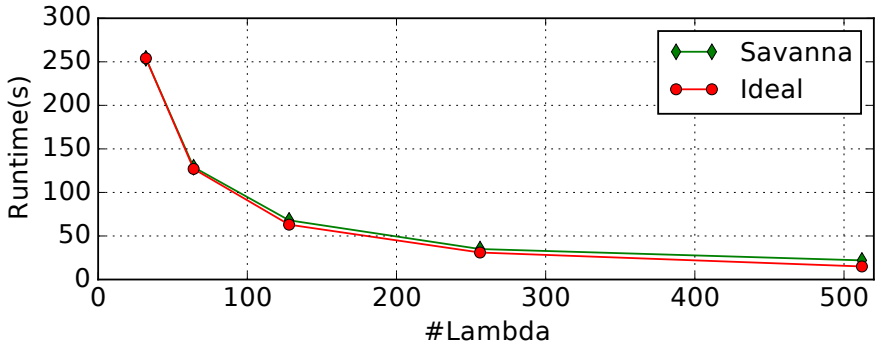


Figure 3.10: Measuring Savanna's strong scaling performance by sorting 512GB data using 32 to 512 concurrent invocations.

GC Scenario	Map	Reduce
GC Disabled	63	77
GC (No `writes` paused)	66	73
GC (With `writes` paused)	62	129

Table 3.3: Running the sort benchmark with different garbage collection scenarios. Only a GC that pauses `writes` impacts performance.

A job that triggers garbage collection. Since garbage collection runs in the background, it does not affect job performance. However, in the rare case where a invocation quickly fills up the local cache, Savanna pauses the `writes` and starts garbage collection. The pause slows the invocation of the named function and we evaluate how the Savanna behaves in this unusual scenario.

We introduce an artificial limit on the storage such that `write` is paused as long as the cache usage hits the artificial limit. We run the sort benchmark and set the artificial limit such that the reduce tasks are paused and garbage collection is executed during the reduce phase. We evaluate three cases here, (a) a normal execution with garbage collection disabled, (b) an execution with background garbage collection, but not pausing the `writes`, and (c) a garbage collection due to a lack of space, with `writes` paused. Table 3.3 shows the runtime of each scenario decomposed into stages. Having background garbage collection has no observable performance impact. However, a garbage collection that pauses the `writes` reduces performance. Despite its lower performance in this uncommon condition, Savanna ensures the correctness and consistency of the named functions and files.

A job whose data is garbage collected. A job's data can be evicted to persistent storage by the garbage collector in the case of low memory. To evaluate the sort workload with varying percentage of shuffle files evicted to S3, we inject some code that forces certain files to be garbage collected even if there is enough memory. Figure 3.11 plots the runtime of sorting 160GB of data with different eviction ratio. Savanna exhibits graceful performance degradation under memory constraints.

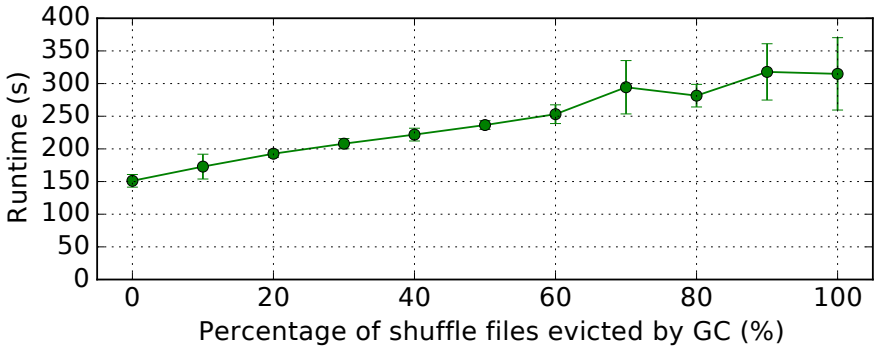


Figure 3.11: Runtime of sorting 160GB data with varying eviction ratio. Savanna's performance degrades gracefully under memory constraints.

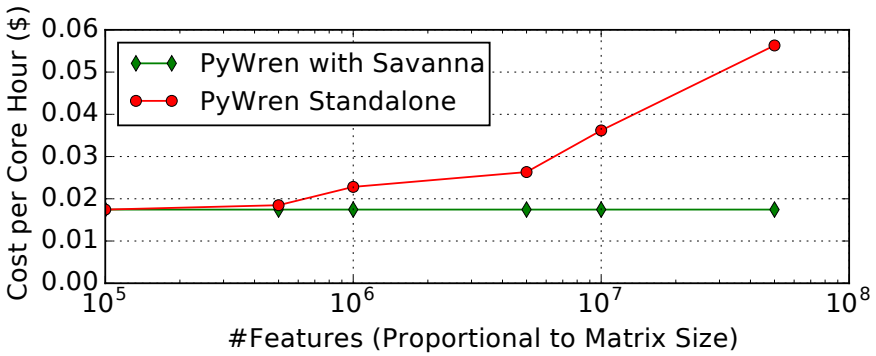


Figure 3.12: Savanna has better scalability per core-hour. By using Savanna, the workload can be executed on smaller and cheaper instances, as Savanna reduces the data shuffle cost.

3.5.6 Case Study: Matrix Multiplication

We describe an implementation of matrix multiplication in PyWren, a serverless framework designed for scientific computing and machine learning tasks. Here we show how Savanna can be seamlessly integrated to provide a significant speedup on this workload. To integrate Savanna we changed roughly 10 lines of code that performed I/O using Boto3. Additionally 30 lines of code were added to PyWren to launch the Savanna metadata coordinator and the Savanna agent.

A distributed matrix multiplication partitions the matrix into chunks and each chunk is multiplied with all other chunks. Therefore, caching is particularly effective in this case as each chunk is read multiple times. Consider a matrix that is partitioned to n chunks, the benefit grows linearly with n, where Savanna saves $(2n-2)$ S3 reads. Further, the PyWren implementation of this workload attempts to minimize shuffles by having large chunks that only fits into more expensive machines. Savanna allows further scale out the computation at low cost, as it avoids expensive S3-based shuffle. The evaluation in Figure 3.12 shows that Savanna reduces the cost per core-hour, especially when the matrix size is large.

3.6 Related Work

Compared with previous open source serverless offerings, such as OpenLambda [64], OpenWhisk [100] and PyWren [75], Savanna is the first attempt to bring together more complex features such as consistency, fault tolerance and caching with one simple and unified file API. We believe our approach is similar to the serverless paradigm itself in spirit, which offers everyone a simple interface to operate the complex cloud. Next, we discuss papers related to our consistency and fault tolerance mechanism.

Consistency. Prior work including work "Bolt-On Consistency" [30], Tapir [144], and others have looked at providing consistency on top of an otherwise inconsistent storage system. On the one hand, many of these works assume no changes to the storage system and provide weaker forms of consistency (*e.g.*, causal consistency) and do not provide atomicity. These systems do not require use of any coordination mechanism (*e.g.*, what is provided by the metadata coordinator in Savanna) and only assume changes to application logic. By contrast Savanna provides both atomicity and a stronger form of consistency (snapshot isolation), but as a result requires use of the metadata coordinator. On the other hand, Tapir implements a strict serializability, a stronger form of consistency, but requires changes to the storage layer – in particular requiring storage to implement consensus mode. This is not implemented by current blob stores and as a result Tapir cannot be implemented on top of blob stores. While Tapir can be implemented on top of services like Spanner (available in GCE), it does not provide any performance benefits in this case. By contrast, Savanna requires no changes to the blob store.

Savanna also makes use of lease locks [59] to ensure that a file cannot be indefinitely locked for writes, even in the presence of network partitions and failures. Leases have previously been used in several file systems including NFSv4 [119] and Ceph [135]. Setting the lease timeout is a challenge in many of these systems since a client may be unable to renew a lease in time due to temporary issues in network connectivity or in the software stack. By contrast, lease timeouts are easily determined in the serverless setting where the maximum execution duration is known in advance.

Fault Tolerance with Lineage. Lineage has been previously used for fault tolerance in a variety of settings including scientific computing [38], databases [41], and data analytics frameworks [74, 142]. However, these previous applications assume prior knowledge about the computation job, in particular they assume knowledge about job dependencies and data lifetimes – this information enables efficient lineage tracking (requiring limited runtime tracking) and storage. In contrast Savanna is designed to be used by arbitrary computation frameworks, and assumes limited visibility into application semantics – in particular Savanna is unaware of data dependencies, and lifetimes prior to the job execution. As a result we rely on runtime instrumentation to collect lineage information, and require data to be durably persisted before pruning lineage information.

3.7 Conclusion

We designed and implemented Savanna, an architectural extension for serverless computing that encapsulates consistency, fault-tolerance and high performance caching inside a simple and unified file API. By evaluating a wide range of applications, we found that Savanna improves application performance between $1.4\times$ and $6.4\times$. Savanna only requires the cloud service providers to modify their platform and it is mostly transparent to the cloud customers. We have ported Savanna to both OpenWhisk and PyWren with little effort, and it can be easily ported to other serverless frameworks. Savanna is in use by machine learning researchers at our institution and we open sourced Savanna on Github for evaluation and future research.

Chapter 4

Network Requirements for Resource Disaggregation

4.1 Introduction

Existing datacenters are built using servers, each of which tightly integrates a small amount of the various resources needed for a computing task (CPU, memory, storage). While such server-centric architectures have been the mainstay for decades, recent efforts suggest a forthcoming paradigm shift towards a *disaggregated* datacenter (DDC), where each resource type is built as a standalone resource "blade" and a network fabric interconnects these resource blades. Examples of this include Facebook Disaggregated Rack [50], HP "The Machine" [126], Intel Rack Scale Architecture [72], SeaMicro [115] as well as prototypes from the computer architecture community [19, 84, 95].

These industrial and academic efforts have been driven largely by hardware architects because CPU, memory and storage technologies exhibit significantly different trends in terms of cost, performance and power scaling [89, 97, 98, 132]. This, in turn, makes it increasingly hard to adopt evolving resource technologies within a server-centric architecture (*e.g.*,, the memory-capacity wall making CPU-memory co-location unsustainable [139]). By decoupling these resources, DDC makes it easier for each resource technology to evolve independently and reduces the time-to-adoption by avoiding the burdensome process of redoing integration and motherboard design.[1] In addition, disaggregation also enables fine-grained and efficient provisioning and scheduling of individual resources across jobs [63].

A key enabling (or blocking) factor for disaggregation will be the network, since disaggregating CPU from memory and disk requires that the inter-resource communication that used to be contained *within* a server must now traverse the network fabric. Thus, to support good application-level performance it becomes critical that the network fabric provide low latency communication

[1] We assume partial CPU-memory disaggregation, where each CPU has some local memory. We believe this is a reasonable intermediate step toward full CPU-memory disaggregation.

for this increased load. It is perhaps not surprising then that prototypes from the hardware community [19, 50, 72, 84, 95, 115, 126] all rely on new high-speed network components – e.g., silicon photonic switches and links, PCIe switches and links, new interconnect fabrics, etc. The problem, however, is that these new technologies are still a long way from matching existing commodity solutions with respect to cost efficiency, manufacturing pipelines, support tools, and so forth. Hence, at first glance, disaggregation would appear to be gated on the widespread availability of new networking technologies.

But are these new technologies strictly *necessary* for disaggregation? Somewhat surprisingly, despite the many efforts towards and benefits of resource disaggregation, there has been little systematic evaluation of the network requirements for disaggregation. In this paper, we take a first stab at evaluating the *minimum* (bandwidth and latency) requirements that the network in disaggregated datacenters must provide. We define the minimum requirement for the network as that which allows us to maintain application-level performance close to server-centric architectures; i.e., at minimum, we aim for a network that keeps performance degradation small for current applications while still enabling the aforementioned qualitative benefits of resource disaggregation.

Using a combination of emulation, simulation, and implementation, we evaluate these minimum network requirements in the context of ten workloads spanning seven popular open-source systems — Hadoop, Spark, GraphLab, Timely dataflow [92,128], Spark Streaming, memcached [88], HERD [77], and SparkSQL. We focus on current applications such as the above because, as we elaborate in §4.3, they represent the worst case in terms of the application *degradation* that may result from disaggregation.

Our key findings are:

- Network bandwidth in the range of $40-100$Gbps is sufficient to maintain application-level performance within 5% of that in existing datacenters; this is easily in reach of existing switch and NIC hardware.
- Network latency in the range of $3-5\mu$s is needed to maintain application-level performance. This is a challenging task. Our analysis suggests that the primary latency bottleneck stems from network software rather than hardware: we find the latency introduced by the endpoint is roughly 66% of the inter-rack latency and roughly 81% of the intra-rack latency. Thus many of the switch hardware optimizations (such as terabit links) pursued today can optimize only a small fraction of the overall latency budget. Instead, work on bypassing the kernel for packet processing and NIC integration [37] could significantly impact the feasibility of resource disaggregation.
- We show that the root cause of the above bandwidth and latency requirements is the application's memory bandwidth demand.
- While most efforts focus on disaggregating at the rack scale, our results show that for some applications, disaggregation at the datacenter scale is feasible.

- Finally, our study shows that transport protocols frequently deployed in today's datacenters (TCP or DCTCP) fail to meet our target requirements for low latency communication with the DDC workloads. However, some recent research proposals [16, 55] do provide the necessary end-to-end latencies.

Taken together, our study suggests that resource disaggregation need not be gated on the availability of new networking hardware: instead, minimal performance degradation can be achieved with existing network hardware (either commodity, or available shortly).

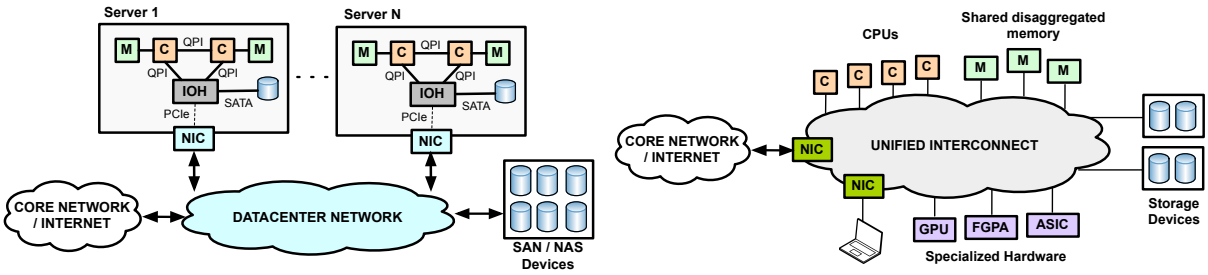
There are two important caveats to this. First, while we may not need network changes, we will need changes in hosts, for which RDMA and NIC integration (for hardware) and pFabric or pHost (for transport protocols) are promising directions. Second, our point is not that new networking technologies are not worth pursuing but that the adoption of disaggregation *need not be coupled* to the deployment of these new technologies. Instead, early efforts at disaggregation can begin with existing network technologies; system builders can incorporate the newer technologies when doing so makes sense from a performance, cost, and power standpoint.

Before continuing, we note three limitations of our work. First, our results are based on ten specific workloads spanning seven open-source systems with varying designs; we leave to future work an evaluation of whether our results generalize to other systems and workloads.[2] Second, we focus primarily on questions of network design for disaggregation, ignoring many other systems questions (*e.g.*,, scheduler designs or software stack) modulo discussion on understanding latency bottlenecks. However, if the latter does turn out to be the more critical bottleneck for disaggregation, one might view our study as exploring whether the network can "get out of the way" (as often advocated [60]) even under disaggregation. Finally, our work looks ahead to an overall system that does not yet exist and hence we must make assumptions on certain fronts (e.g., hardware design and organization, data layout, etc.). We make what we believe are sensible choices, state these choices explicitly in §4.2, and to whatever extent possible, evaluate the sensitivity of these choices on our results. Nonetheless, our results are dependent on these choices, and more experience is needed to confirm their validity.

4.2 Disaggregated Datacenters

Figure 4.1 illustrates the high-level idea behind a disaggregated datacenter. A DDC comprises standalone hardware "blades" for each resource type, interconnected by a network fabric. Multiple prototypes of disaggregated hardware already exist — Intel RSA [72], HP "The Machine" [126], Facebook's Disaggregated Rack [50], Huawei's DC3.0 [67], and SeaMicro [115], as well as research prototypes like FireBox [19], soNUMA [95], and memory blades [84]. Many of these systems are proprietary and/or in the early stages of development; nonetheless, in our study we

[2]We encourage other researchers to extend the evaluation with our emulator. https://github.com/NetSys/disaggregation



(a) Current datacenter (b) Disaggregated datacenter

Figure 4.1: High-level architectural differences between server-centric and resource-disaggregated datacenters.

Communication	Latency (ns)	Bandwidth (Gbps)
CPU – CPU	10	500
CPU – Memory	20	500
CPU – Disk (SSD)	10^4	5
CPU – Disk (HDD)	10^6	1

Table 4.1: Typical latency and peak bandwidth requirements within a traditional server. Numbers vary between hardware.

draw from what information is publicly available to both borrow from and critically explore the design choices made by existing hardware prototypes.

In this section, we present our assumptions regarding the hardware (§4.2.1) and system (§4.2.2) architecture in a disaggregated datacenter. We close the section by summarizing the key open design choices that remain after our assumptions (§4.2.3); we treat these as design "knobs" in our evaluation.

4.2.1 Assumptions: Hardware Architecture

Partial CPU-memory disaggregation. In general, disaggregation suggests that each blade contains one particular resource with a direct interface to the network fabric (Fig. 4.1). One exception to this strict decoupling is CPU blades: each CPU blade retains some amount of *local* memory that acts as a cache for remote memory dedicated for cores on that blade[3]. Thus, CPU-memory disaggregation can be viewed as expanding the memory hierarchy to include a remote level, which all CPU blades share.

This architectural choice is reported in prior work [19,67,84,85]. While we assume that partial CPU-memory disaggregation will be the norm, we go a step further and evaluate how the amount

[3]We use "remote memory" to refer to the memory located on a standalone memory blade.

Class	Application Domain	Application	System	Dataset
Class A	Off-disk Batch	WordCount	Hadoop	Wikipedia edit history [136]
	Off-disk Batch	Sort	Hadoop	Sort benchmark generator
	Graph Processing	Collaborative Filtering	GraphLab	Netflix movie rating data [94]
	Point Queries	Key-value store	Memcached	YCSB
	Streaming Queries	Stream WordCount	Spark Streaming	Wikipedia edit history [136]
Class B	In-memory Batch	WordCount	Spark	Wikipedia edit history [136]
	In-memory Batch	Sort	Spark	Sort benchmark generator
	Parallel Dataflow	Pagerank	Timely Dataflow	Friendster Social Network [53]
	In-memory Batch	SQL	Spark SQL	Big Data Benchmark [33]
	Point Queries	Key-value store	HERD	YCSB

Table 4.2: Applications, workloads, systems and datasets used in our study. We stratify the classes in Section 4.3.

of local memory impacts *network* requirements in terms of network bandwidth and latency, and transport-layer flow completion times.

Cache coherence domain is limited to a single compute blade. As articulated by others [19, 67, 126], this has the important implication that CPU-to-CPU cache coherence traffic does not hit the network fabric. While partial CPU-memory disaggregation reduces the traffic hitting the network, cache coherence traffic can not be cached and hence directly impacts the network. This assumption is necessary because an external network fabric is unlikely to support the latency and bandwidth requirements for inter-CPU cache coherence (Table 4.1).

Resource Virtualization. Each resource blade must support virtualization of its resources; this is necessary for resources to be logically aggregated into higher-level abstractions such as VMs or containers. Virtualization of IO resources is widely available even today: many IO device controllers now support virtualization via PCIe, SR-IOV, or MR-IOV features [9] and the same can be leveraged to virtualize IO resources in DDC. The disaggregated memory blade prototyped by Lim et al. [84] includes a controller ASIC on each blade that implements address translation between a remote CPU's view of its address space and the addressing used internally within the blade. Other research efforts assume similar designs. We note that while the implementation of such blades may require some additional new hardware, it requires no change to existing components such as CPUs, memory modules, or storage devices themselves.

Scope of disaggregation. Existing prototypes limit the scope of disaggregation to a very small number of racks. For example, FireBox [19] envisions a single system as spanning approximately three racks and assumes that the *logical* aggregation and allocation of resources is similarly scoped; i.e., the resources allocated to a higher-level abstraction such as a VM or a container are selected from a single FireBox. Similarly, the scope of disaggregation in Intel's RSA is a single rack [72]. In contrast, in a hypothetical datacenter-scale disaggregated system, resources assigned to (for example) a single VM could be selected from anywhere in the datacenter.

Network designs. Corresponding to their assumed scope of disaggregation, existing prototypes assume a different network architecture for within the rack(s) that form a unit of disaggregation vs. between such racks. To our knowledge, all existing DDC prototypes use specialized – even proprietary [67, 72, 115] – network technologies and protocols within a disaggregated rack(s). For example, SeaMicro uses a proprietary Torus-based topology and routing protocol within its disaggregated system; Huawei propose a PCIe-based fabric [96]; FireBox assumes an intra-FireBox network of 1Tbps Silicon photonic links interconnected by high-radix switches [19,78]; and Intel's RSA likewise explores the use of Silicon photonic links and switches.

Rather than simply accepting the last two design choices (rack-scale disaggregation and specialized network designs), we critically explore when and why these choices are necessary. Our rationale in this is twofold. First, these are both choices that appear to be motivated not by fundamental constraints around disaggregating memory or CPU at the hardware level, but rather by the assumption that existing networking solutions cannot meet the (bandwidth/latency) requirements that disaggregation imposes on the network. To our knowledge, however, there has been no published evaluation showing this to be the case; hence, we seek to develop quantifiable arguments that either confirm or refute the need for these choices.

Second, these choices are likely to complicate or delay the deployment of DDC. The use of a different network architecture within vs. between disaggregated islands leads to the complexity of a two-tier heterogeneous network architecture with different protocols, configuration APIs, etc., for each; e.g., in the context of their FireBox system, the authors envisage the use of special gateway devices that translate between their custom intra-FireBox protocols and TCP/IP that is used between FireBox systems; Huawei's DC3.0 makes similar assumptions. Likewise, many of the specialized technologies these systems use (e.g., Si-photonic [124]) are still far from mainstream. Hence, once again, rather than assume change is necessary, we evaluate the possibility of maintaining a uniform "flat" network architecture based on existing commodity components as advocated in prior work [12, 61, 62].

4.2.2 Assumptions: System Architecture

In contrast to our assumptions regarding hardware which we based on existing prototypes, we have less to guide us on the systems front. We thus make the following assumptions, which we believe are reasonable:

System abstractions for *logical* resource aggregations. In a DDC, we will need system abstractions that represent a logical aggregation of resources, in terms of which we implement resource allocation and scheduling. One such abstraction in existing datacenters is a VM: operators provision VMs to aggregate slices of hardware resources within a server, and schedulers place jobs across VMs. While not strictly necessary, we note that the VM model can still be useful in DDC.[4] For convenience, in this paper we assume that computational resources are still aggregated to form

[4]In particular, continuing with the abstraction of a VM would allow existing software infrastructure — i.e., hypervisors, operating systems, datacenter middleware, and applications — to be reused with little or no modification.

VMs (or VM-like constructs), although now the resources assigned to a VM come from distributed hardware blades. Given a VM (or VM-like) abstraction, we assign resources to VMs differently based on the *scope* of disaggregation that we assume: for rack-scale disaggregation, a VM is assigned resources from within a single rack while, for datacenter-scale disaggregation, a VM is assigned resources from anywhere in the datacenter.

Hardware organization. We assume that resources are organized in racks as in today's datacenters. We assume a "mixed" organization in which each rack hosts a mix of different types of resource blades, as opposed to a "segregated" organization in which a rack is populated with a single type of resource (e.g., all memory blades). This leads to a more uniform communication pattern which should simplify network design and also permits optimizations that aim to localize communication; e.g., co-locating a VM within a rack, which would not be possible with a segregated organization.

Page-level remote memory access. In traditional servers, the typical memory access between CPU and DRAM occurs in the unit of a cache-line size (64B in x86). In contrast, we assume that CPU blades access remote memory at the granularity of a page (4KB in x86), since page-level access has been shown to better exploit spatial locality in common memory access patterns [84]. Moreover, this requires little or no modification to the virtual memory subsystems of hypervisors or operating systems, and is completely transparent to user-level applications.

Block-level distributed data placement. We assume that applications in DDC read and write large files at the granularity of "sectors" (512B in x86). Furthermore, the disk block address space is range partitioned into "blocks", that are uniformly distributed across the disk blades. The latter is partially motivated by existing distributed file systems (*e.g.*,, HDFS) and also enables better load balancing.

4.2.3 Design knobs

Given the above assumptions, we are left with two key system design choices that we treat as "knobs" in our study: *the amount of local memory on compute blades* and *the scope of disaggregation* (e.g., rack- or datacenter-scale). We explore how varying these knobs impacts the network requirements and traffic characteristics in DDC in the following section.

The remainder of this paper is organized as follows. We first analyze network-layer bandwidth and latency requirements in DDC (§4.3) *without* considering contention between network flows, then in §4.4 relax this constraint. We end with a discussion of the future directions in §4.5.

4.3 Network Requirements

We start by evaluating network latency and bandwidth requirements for disaggregation. We describe our evaluation methodology (§4.3.1), present our results (§4.3.2) and then discuss their

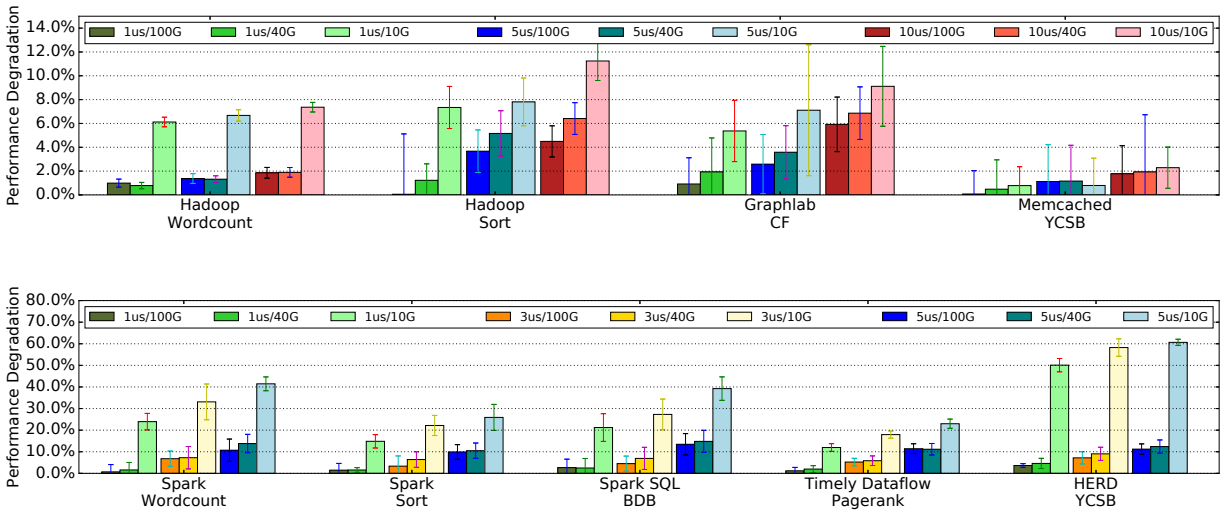


Figure 4.2: Comparison of application-level performance in disaggregated datacenters with respect to existing server-centric architectures for different latency/bandwidth configurations and 25% local memory on CPU blades — Class A apps (top) and Class B apps (bottom). To maintain application-level performance within reasonable performance bounds ($\sim$ 5% on an average), Class A apps require 5μs end-to-end latency and 40Gbps bandwidth, and Class B apps require 3μs end-to-end latency and $40-100$Gbps bandwidth. See §4.3.2 for detailed discussion.

implications (§4.3.3).

4.3.1 Methodology

In DDC, traffic between resources that was contained within a server is now carried on the "external" network. As with other types of interconnects, the key requirement will be low latency and high throughput to enable this disaggregation. We review the forms of communication between resources within a server in Table 4.1 to examine the feasibility of such a network. As mentioned in §4.2, CPU-to-CPU cache coherence traffic does not cross the external network. For I/O traffic to storage devices, the current latency and bandwidth requirements are such that we can expect to consolidate them into the network fabric with low performance impact, assuming we have a 40Gbps or 100Gbps network. Thus, the dominant impact to application performance will come from CPU-memory disaggregation; hence, we focus on evaluating the network bandwidth and latency required to support remote memory.

As mentioned earlier, we assume that remote memory is managed at the page granularity, in conjunction with virtual memory page replacement algorithms implemented by the hypervisor or operating system. For each paging operation there are two main sources of performance penalty: i) the software overhead for trap and page eviction and ii) the time to transfer pages over the network. Given our focus on network requirements, we only consider the latter in this paper (modulo a brief

discussion on current software overheads later in this section).

Applications. We use workloads from diverse applications running on real-world and benchmark datasets, as shown in Table 4.2. The workloads can be classified into two classes based on their performance characteristics. We elaborate briefly on our choice to take these applications as is, rather than seek to optimize them for DDC. Our focus in this paper is on understanding whether and why networking might gate the deployment of DDC. For this, we are interested in the degradation that applications might suffer if they were to run in DDC. We thus compare the performance of an application in a server-centric architecture to its performance in the disaggregated context we consider here (with its level of bandwidth and local memory). This would be strictly worse than if we compared to the application's performance if it had been rewritten for this disaggregated context. Thus, legacy (i.e., server-centric) applications represent the worst-case in terms of potential degradation and give us a lower bound on the network requirements needed for disaggregation (it might be that rewritten applications could make do with lower bandwidths). Clearly, if new networking technologies exceed this lower bound, then all applications (legacy and "native" DDC) will benefit. Similarly, new programming models designed to exploit disaggregation can only improve the performance of all applications. The question of how to achieve improved performance through new technologies and programming models is an interesting one but beyond the scope of our effort and hence one we leave to future work.

Emulating remote memory. We run the following applications unmodified with 8 threads and reduce the amount of local memory directly accessible by the applications. To emulate remote memory accesses, we implement a special swap device backed by the remaining physical memory rather than disk. This effectively partitions main memory into "local" and "remote" portions where existing page replacement algorithms control when and how pages are transferred between the two. We tune the amount of "remote" memory by configuring the size of the swap device; remaining memory is "local". We intercept all page faults and inject artificial delays to emulate network round-trip latency and bandwidth for each paging operation. Note that when a page fault occurs, the page is not actually swapped over the network; instead, it is swapped to the remaining part of the memory on the same machine.

We measure relative application-level performance on the basis of job completion time as compared to the zero-delay case. Thus, our results do not account for the delay introduced by software overheads for page operations and should be interpreted as *relative* performance degradations over different network configurations. Note too that the delay we inject is purely an artificial parameter and hence does not (for example) realistically model queuing delays that may result from network congestion caused by the extra traffic due to disaggregation; we consider network-wide traffic and effects such as congestion in §4.4.

Testbed. Each application operates on $\sim$ 125GB of data equally distributed across an Amazon EC2 cluster comprising 5 `m3.2xlarge` servers. Each of these servers has 8 vCPUs, 30GB main memory, $2 \times$ 80GB SSD drives and a 1Gbps access link bandwidth. We enabled EC2's Virtual Private Network (VPC [23]) capability in our cluster to ensure no interference with other Amazon

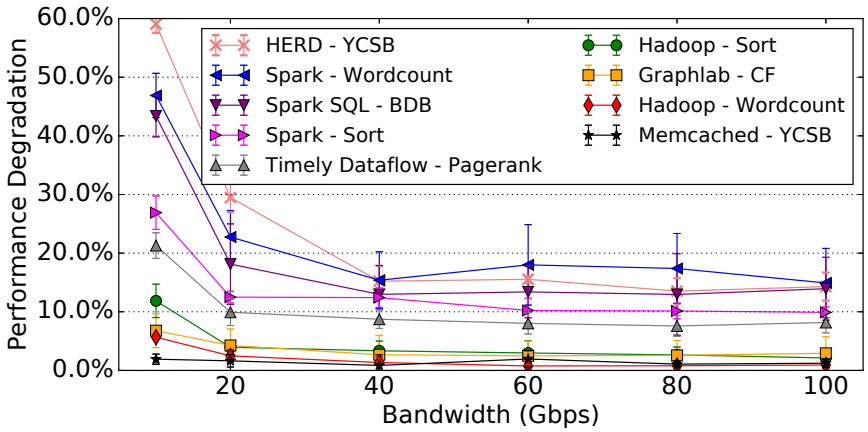


Figure 4.3: Impact of network bandwidth on the results of Figure 4.2 for end-to-end latency fixed to 5μs and local memory fixed to 25%.

Network Provision	Class A	Class B
5μs, 40Gbps	20%	35%
3μs, 100Gbps	15%	30%

Table 4.3: Class B apps require slightly higher local memory than Class A apps to achieve an average performance penalty under 5% for various latency-bandwidth configurations.

EC2 instances.

We verified that `m3.2xlarge` instances' 1Gbps access links were not a bottleneck to our experiment in two ways. First, in all cases where the network approached full utilization, CPU was fully utilized, indicating that the CPU was not blocked on network calls. Next, we ran our testbed on `c3.4xlarge` instances with 2Gbps access links (increased network bandwidth with roughly the same CPU). We verified that even with more bandwidth, all applications for which link utilization was high maintained high CPU utilization. This aligns with the conclusions drawn in [102].

We run batch applications (Spark, Hadoop, Graphlab, and Timely Dataflow) in a cluster with 5 worker nodes and 1 master node; the job request is issued from the master node. For point-query applications (memcached, HERD), requests are sent from client to server across the network. All applications are multi-threaded, with the same number of threads as cores. To compensate for the performance noise on EC2, we run each experiment 10 times and take the median result.

4.3.2 Results

We start by evaluating application performance in a disaggregated vs. a server-centric architecture. Figure 4.2 plots the performance degradation for each application under different assumptions about the latency and bandwidth to remote memory. In these experiments, we set the local memory

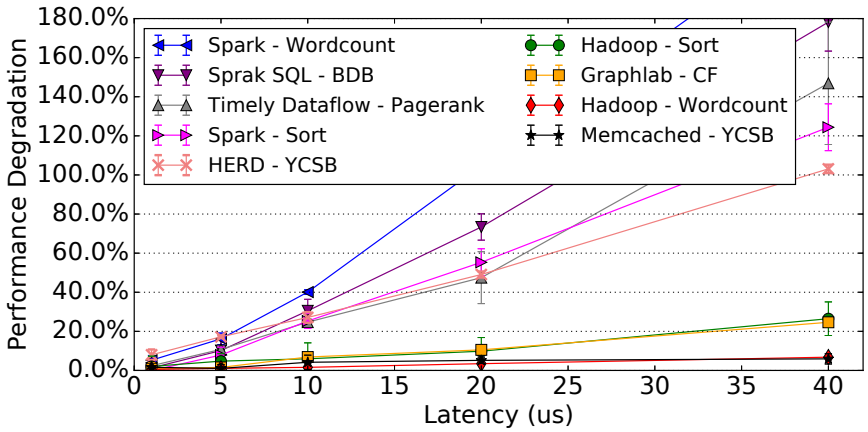


Figure 4.4: Impact of network latency on the results of Figure 4.2 for bandwidth fixed to 40Gbps and local memory fixed to 25%.

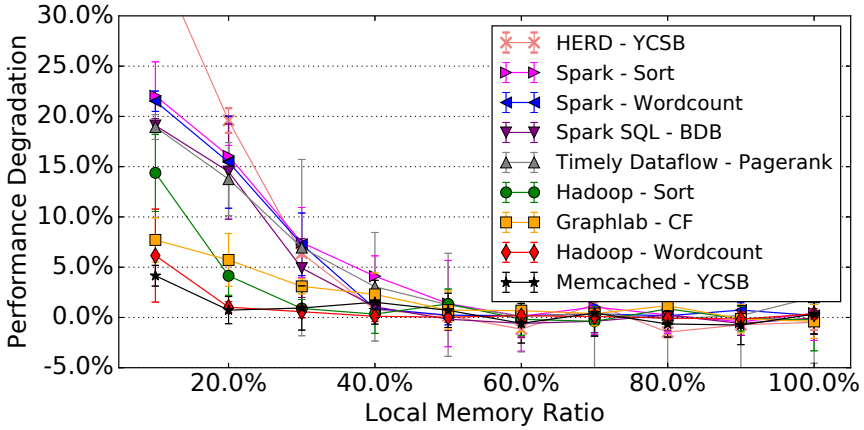
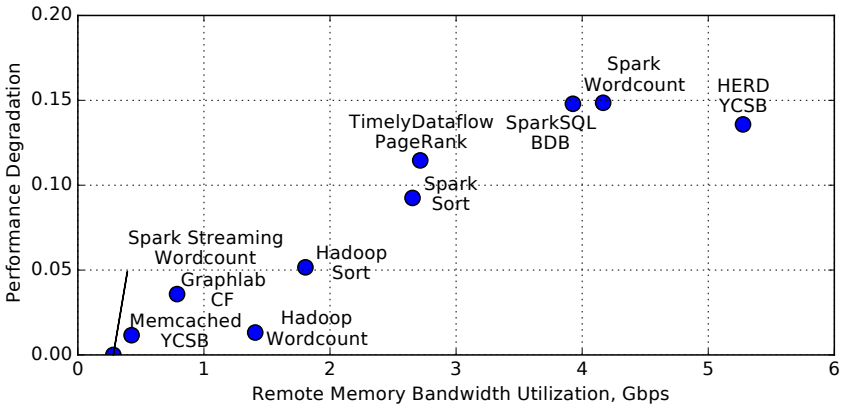


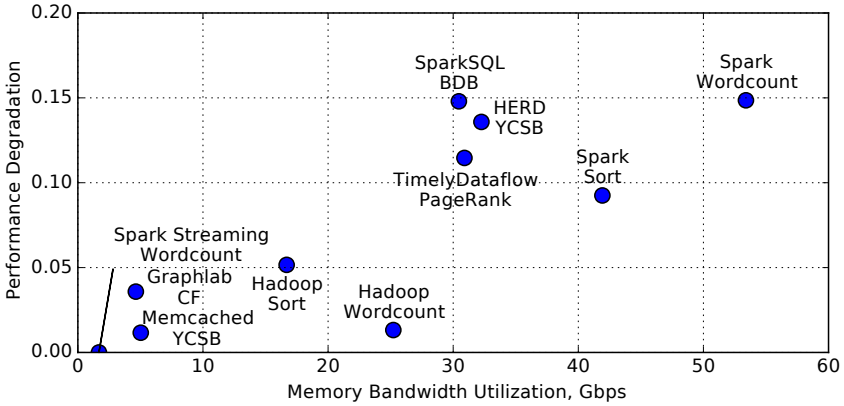
Figure 4.5: Impact of "local memory" on the results of Figure 4.2 for end-to-end latency fixed to 5μs and network bandwidth 40Gbps. Negative values are due to small variations in timings between runs.

in the disaggregated scenario to be 25% of that in the server-centric case (we will examine our choice of 25% shortly). Note that the injected latency is constant across requests; we leave studying the effects of possibly high tail latencies to future work.

From Figure 4.2, we see that our applications can be broadly divided into two categories based on the network latency and bandwidth needed to achieve a low performance penalty. For example, for the applications in Fig. 4.2 (top) — Hadoop Wordcount, Hadoop Sort, Graphlab and Memcached — a network with an end-to-end latency of 5μs and bandwidth of 40Gbps is sufficient to maintain an average performance penalty under 5%. In contrast, the applications in Fig. 4.2 (bottom) — Spark Wordcount, Spark Sort, Timely, SparkSQL BDB, and HERD — require network latencies of 3μs and 40 – 100Gbps bandwidth to maintain an average performance penalty under



(a) Remote Memory Bandwidth Utilization



(b) Memory Bandwidth Utilization

Figure 4.6: Performance degradation of applications is correlated with the swap memory bandwidth and overall memory bandwidth utilization.

8%. We term the former applications *Class A* and the latter *Class B* and examine the feasibility of meeting their respective requirements in §4.3.3. We found that Spark Streaming has a low memory utilization. As a result, its performance degradation is near zero in DDC, and we show it only in Figure 4.6.

Sensitivity analysis. Next, we evaluate the sensitivity of application performance to network bandwidth and latency. Fig. 4.3 plots the performance degradation under increasing network bandwidth assuming a fixed network latency of 5μs while Fig. 4.4 plots degradation under increasing latency for a fixed bandwidth of 40Gbps; in both cases, local memory is set at 25% as before. We see that beyond 40Gbps, increasing network bandwidth offers little improvement in application-level performance. In contrast, performance — particularly for Class B apps — is very sensitive to network latency; very low latencies ($3-5\mu$s) are needed to avoid non-trivial performance degradation.

Finally, we measure how the amount of local memory impacts application performance. Figure 4.5 plots the performance degradation that results as we vary the fraction of local memory from 100% (which corresponds to no CPU-memory disaggregation) down to 10%, assuming a fixed network latency and bandwidth of 5μs and 40Gbps respectively; note that the 25% values (interpolated) in Figure 4.5 correspond to 5μs, 40Gbps results in Figure 4.2. As expected, we see that Class B applications are more sensitive to the amount of local memory than Class A apps; e.g., increasing the amount of local memory from 20% to 30% roughly halves the performance degradation in Class B from approximately 15% to 7%. In all cases, increasing the amount of local memory beyond 40% has little to no impact on performance degradation.

Understanding (and extrapolating from) our results. One might ask *why* we see the above requirements – i.e., what characteristic of the applications we evaluated led to the specific bandwidth and latency requirements we report? An understanding of these characteristics could also allow us to generalize our findings to other applications.

We partially answer this question using Figure 4.6, which plots the performance degradation of the above nine workloads against their swap and memory bandwidth[5]. Figure 4.6a and 4.6b show that an application's performance degradation is very strongly correlated with its swap bandwidth and well correlated with its memory bandwidth. The clear correlation with swap bandwidth is to be expected. That the overall memory bandwidth is also well correlated with the resultant degradation is perhaps less obvious and an encouraging result as it suggests that an application's memory bandwidth requirements might serve as a rough indicator of its expected degradation under disaggregation: this is convenient as memory bandwidth is easily measured without requiring any of our instrumentation (i.e., emulating remote memory by a special swap device, etc.). Thus it should be easy for application developers to get a rough sense of the performance degradation they might expect under disaggregation and hence the urgency of rewriting their application for disaggregated contexts.

We also note that there is room for more accurate predictors: the difference between the two figures (Figs. 4.6a and 4.6b) shows that the locality in memory access patterns does play some role in the expected degradation (since the swap bandwidth which is a better predictor captures only the subset of memory accesses that miss in local memory). Building better prediction models that account for an application's memory access pattern is an interesting question that we leave to future work.

Access Granularity. Tuning the granularity of remote memory access is an interesting area for future work. For example, soNUMA [95] accesses remote memory at cache-line size granularity, which is much smaller than page-size. This may allow point-query applications to optimize their

[5]We use Intel's Performance Counter Monitor software [71] to read the uncore performance counters that measure the number of bytes written to and read from the integrated memory controller on each CPU. We confirmed using benchmarks designed to saturate memory bandwidth [145] that we could observe memory bandwidth utilization numbers approaching the reported theoretical maximum. As further validation, we verified that our Spark SQL measurement is consistent with prior work [108].

Component	Baseline (μs)		With RDMA (μs)		With RDMA + NIC Integr. (μs)	
	Inter-rack	Intra-rack	Inter-rack	Intra-rack	Inter-rack	Intra-rack
OS	2×0.95	2×0.95	0	0	0	0
Data copy	2×1.00	2×1.00	2×1.00	2×1.00	2×0.50	2×0.50
Switching	6×0.24	2×0.24	6×0.24	2×0.24	6×0.24	2×0.24
Propagation (Inter-rack)	4×0.20	0	4×0.20	0	4×0.20	0
Propagation (Intra-rack)	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02	4×0.02
Transmission	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82	1×0.82
Total	7.04μs	5.28μs	5.14μs	3.38μs	4.14μs	2.38μs

Table 4.4: Achievable round-trip latency (Total) and the components that contribute to the round-trip latency (see discussion in §4.3.3) on a network with 40Gbps access link bandwidth (one can further reduce the **Total** by 0.5μs using 100Gbps access link bandwidth). The baseline denotes the latency achievable with existing network technology. The fractional part in each cell is the latency for one traversal of the corresponding component and the integral part is the number of traversal performed in one round-trip time (see discussion in §4.3.3).

dependence on remote memory. On the other hand, developers of applications which use large, contiguous blocks of memory may wish to use hugepages to reduce the number of page table queries and thus speed up virtual memory mapping. Since Linux currently limits (non-transparent) hugepages from being swapped out of physical memory, exploring this design option is not currently feasible.

Overall, we anticipate that programmers in DDC will face a tradeoff in optimizing their applications for disaggregation depending on its memory access patterns.

Remote SSD and NVM. Our methodology is not limited to swapping to remote memory. In fact, as long as the 3μs latency target is met, there is no limitation on the media of the remote storage. We envision that the remote memory could be replaced by SSD or forthcoming Non-Volatile Memory (NVM) technologies, and anticipate different price and performance tradeoff for these technologies.

Summary of results. In summary, supporting memory disaggregation while maintaining application-level performance within reasonable bounds imposes certain requirements on the network in terms of the end-to-end latency and bandwidth it must provide. Moreover, these requirements are closely related to the amount of local memory available to CPU blades. Table 4.3 summarizes these requirements for the applications we studied. We specifically investigate a few combinations of network latency, bandwidth, and the amount of local memory needed to maintain a performance degradation under 5%. We highlight these design points because they represent what we consider to be sweet spots in achievable targets both for the amount of local memory and for network requirements, as we discuss next.

4.3.3 Implications and Feasibility

We now examine the feasibility of meeting the requirements identified above.

Local memory. We start with the requirement of between $20-30\%$ local memory. In our experiments, this corresponds to between $1.50-2.25$GB/core. We look to existing hardware prototypes for validation of this requirement. The FireBox prototype targets 128GB of local memory shared by 100 cores leading to 1.28GB/core,[6] while the analysis in [84] uses 1.5GB/core. Further, [85] also indicates 25% local memory as a desirable setting, and HP's "The Machine" [127] uses an even larger fraction of local memory: 87%. Thus we conclude that our requirement on local memory is compatible with demonstrated hardware prototypes. Next, we examine the feasibility of meeting our targets for network bandwidth and latency.

Network bandwidth. Our bandwidth requirements are easily met: 40Gbps is available today in commodity datacenter switches *and* server NICs [140]; in fact, even 100Gbps switches and NICs are available, though not as widely [42]. Thus, ignoring the potential effects of congestion (which we consider next in §4.4), providing the network bandwidth needed for disaggregation should pose no problem. Moreover, this should continue to be the case in the future because the trend in link bandwidths currently exceeds that in number of cores [36, 49, 70].

Network latency. The picture is less clear with respect to latency. In what follows, we consider the various components of network latency and whether they can be accommodated in our target budget of 3μs (for Class B apps) to 5μs (for Class A apps).

Table 4.4 lists the six components of the end-to-end latency incurred when fetching a 4KB page using 40Gbps links, together with our estimates for each. Our estimates are based on the following common assumptions about existing datacenter networks: (1) the one-way path between servers in different racks crosses three switches (two ToR and one fabric switch) while that between servers in the same rack crosses a single ToR switch, (2) inter-rack distances of 40m and intra-rack distances of 4m with a propagation speed of 5ns/m, (3) cut-through switches.[7] With this, our round-trip latency includes the software overheads associated with moving the page to/from the NIC at both the sending and receiving endpoints (hence 2x the OS and data copy overheads), 6 switch traversals, 4 link traversals in each direction including two intra-rack and two cross-rack, and the transmission time for a 4KB page (we ignore transmission time for the page request), leading to the estimates in Table 4.4.

We start by observing that the network introduces three unavoidable latency overheads: (i) the data transmission time, (ii) the propagation delay; and (iii) the switching delay. Together, these components contribute to roughly 3.14μs across racks and 1.38μs within a rack.[8]

[6] We thank Krste Asanović for clarification on FireBox's technical specs.

[7] As before, we ignore the queuing delays that may result from congestion at switches – we will account for this in §4.4.

[8] Discussions with switch vendors revealed that they are approaching the fundamental limits in reducing switching delays (for electronic switches), hence we treat the switching delay as unavoidable.

In contrast, the network software at the endpoints is a significant contributor to the end-to-end latency! Recent work reports a round-trip kernel processing time of 950 ns measured on a 2.93GHz Intel CPU running FreeBSD (see [110] for details), while [101] reports an overhead of around 1μs to copy data between memory and the NIC. With these estimates, the network software contributes roughly 3.9μs latency — this represents 55% of the end-to-end latency in our baseline inter-rack scenario and 73% in our baseline intra-rack scenario.

The end-to-end latencies we estimated in our baseline scenarios (whether inter- or intra-rack) fail to meet our target latencies for either Class B or Class A applications. Hence, we consider potential optimizations and technologies that can reduce these latencies. Two technologies show promise: RDMA and integrated NICs.

Using RDMA. RDMA effectively bypasses the packet processing in the kernel, thus eliminating the OS overheads from Table 4.4. Thus, using RDMA (Infiniband [69] or Omnipath [99]), we estimate a reduced end-to-end latency of 5.14μs across racks (column #4 in Table 4.4) and 3.38μs within a rack.

Using NIC integration. Recent industry efforts pursue the integration of NIC functions closer to the CPU [37] which would reduce the overheads associated with copying data to/from the NIC. Rosenblum *et al.* [111] estimate that such integration together with certain software optimizations can reduce copy overheads to sub-microseconds, which we estimate at 0.5μs (similar to [111]).

Using RDMA and NIC integration. As shown in column #5 in Table 4.4, the use of RDMA together with NIC integration reduces the end-to-end latency to 4.14μs across racks; within a rack, this further reduces down to 2.38μs (using the same differences as in column #2 and column #3).

Takeaways. We highlight a few takeaways from our analysis:

- The overhead of network *software* is the key barrier to realizing disaggregation with current networking technologies. Technologies such as RDMA and integrated NICs that eliminate some of these overheads offer promise: reducing end-to-end latencies to 4.14μs between racks and 2.38μs within a rack. However, demonstrating such latencies in a working prototype remains an important topic for future exploration.
- Even assuming RDMA and NIC integration, the end-to-end latency across racks (4.14μs) meets our target latency only for Class A, but not Class B, applications. Our target latency for Class B apps is only met by the end-to-end latency within a rack. Thus, Class B jobs will have to be scheduled within a single rack (or nearby racks). That is, while Class A jobs can be scheduled at blades distributed across the datacenter, Class B jobs will need to be scheduled within a rack. The design and evaluation of such schedulers remains an open topic for future research.
- While new network hardware such as high-bandwidth links (e.g., 100Gbps or even 1Tbps as in [19, 78]) and high-radix switches (*e.g.*,, 1000 radix switch [19]) are certainly useful, they

optimize a relatively small piece of the overall latency in our baseline scenario technologies. All-optical switches also fall into this category – providing both potentially negligible switching delay and high bandwidth. That said, once we assume the benefits of RDMA and NIC integration, then the contribution of new links and switches could bring even the cross-rack latency to within our 3μs target for Class B applications, enabling true datacenter-scale disaggregation; e.g., using 100Gbps links reduces the end-to-end latency to 3.59μs between racks, extremely close to our 3μs.

- Finally, we note that managing network congestion to achieve zero or close-to-zero queuing within the network will be essential; e.g., a packet that is delayed such that it is queued behind (say) 4 packets will accumulate an additional delay of $4 \times 0.82\mu$s! Indeed, reducing such transmission delays may be the reason to adopt high-speed links. We evaluate the impact of network congestion in the following section.

4.4 Network Designs for Disaggregation

Our evaluation has so far ignored the impact of queuing delay on end-to-end latency and hence application performance; we remedy the omission in this section. The challenge is that queuing delay is a function of the overall network design, including: the traffic workload, network topology and routing, and the end-to-end transport protocol. Our evaluation focuses on existing proposals for transport protocols, with standard assumptions about the datacenter topology and routing. However, the input traffic workload in DDC will be very different from that in a server-centric datacenter and, to our knowledge, no models exist that characterize traffic in a DDC.

We thus start by devising a methodology that extends our experimental setup to generate an application-driven input traffic workload (§4.4.1), then describe how we use this traffic model to evaluate the impact of queuing delay (§4.4.2). Finally, we present our results on: (i) how existing transport designs perform under DDC traffic workloads (§4.4.3), and (ii) how existing transport designs impact end-to-end application performance (§4.4.4). To our knowledge, our results represent the first evaluation of transport protocols for DDC.

4.4.1 Methodology: DDC Traffic Workloads

Using our experimental setup from §4.3.1, we collect a remote memory access trace from our instrumentation tool as described in §4.3.1, a network access trace using `tcpdump` [125], and a disk access trace using `blktrace`.

We translate the accesses from the above traces to network flows in our simulated disaggregated cluster by splitting each node into one compute, one memory, and one disk blade and assigning memory blades to virtual nodes.

All memory and disk accesses captured above are associated with a specific address in the corresponding CPU's global virtual address space. We assume this address space is uniformly par-

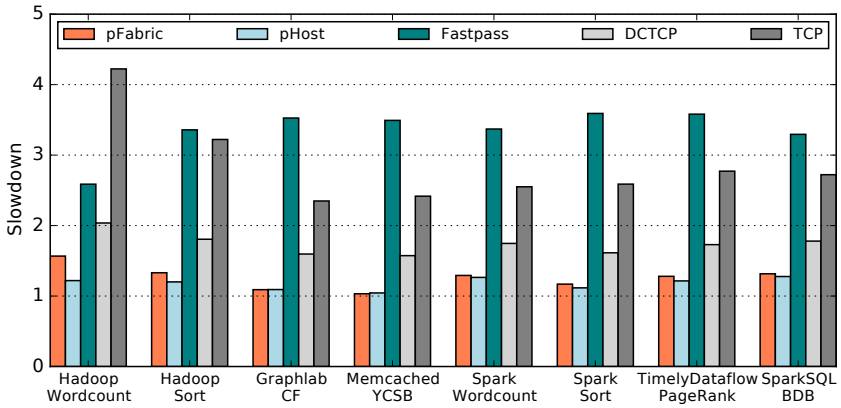


Figure 4.7: The performance of the five protocols for the case of 100Gbps access link capacity. The results for 40Gbps access links lead to similar conclusions. See §4.4.3 for discussion on these results.

titioned across all memory and disk blades reflecting our assumption of distributed data placement (§4.2.2).

One subtlety remains. Consider the disk accesses at a server A in the original cluster: one might view all these disk accesses as corresponding to a flow between the compute and disk blades corresponding to A, but in reality A's CPU may have issued some of these disk accesses in response to a request from a remote server B (*e.g.*,, due to a shuffle request). In the disaggregated cluster, this access should be treated as a network flow between B's compute blade and A's disk blade.

To correctly attribute accesses to the CPU that originates the request, we match network and disk traces across the cluster – e.g., matching the network traffic between B and A to the disk traffic at A – using a heuristic based on both the timestamps and volume of data transferred. If a locally captured memory or disk access request matches a local flow in our `tcpdump` traces, then it is assumed to be part of a remote read and is attributed to the remote endpoint of the network flow. Otherwise, the memory/disk access is assumed to have originated from the local CPU.

4.4.2 Methodology: Queuing delay

We evaluate the use of existing network designs for DDC in two steps. First, we evaluate how existing network designs fare under DDC traffic workloads. For this, we consider a suite of state-of-the-art network designs and use simulation to evaluate their network-layer performance – measured in terms of flow completion time (FCT) – under the traffic workloads we generate as above. We then return to actual execution of our applications (Table 4.2) and once again emulate disaggregation by injecting latencies for page misses. However, now we inject the flow completion times obtained from our best-performing network design (as opposed to the constant latencies from §4.3). This last step effectively "closes the loop", allowing us to evaluate the impact of disaggregation on

application-level performance for realistic network designs and conditions.

Simulation Setup. We use the same simulation setup as prior work on datacenter transports [15, 16, 55]. We simulate a topology with 9 racks (with 144 total endpoints) and a full bisection bandwidth Clos topology with 36KB buffers per port; our two changes from prior work are to use 40Gbps or 100Gbps access links (as per §4.3), and setting propagation and switching delays as discussed in §4.3.3 (Table 4.4 with RDMA and NIC integration). We map the 5 EC2-node cluster into a disaggregated cluster with 15 blades: 5 each of compute, memory and disk. Then, we extract the flow size and inter-arrival time distribution for each endpoint pair in the 15 blades disaggregated cluster, and generate traffic using the distributions. Finally, we embed the multiple disaggregated clusters into the 144-endpoint datacenter with both rack-scale and datacenter-scale disaggregation, where communicating nodes are constrained to be within a rack and unconstrained, respectively.

We evaluate five protocols; in each case, we set protocol-specific parameters following the default settings but adapted to our bandwidth-delay product as recommended.

1. **TCP**, with an initial congestion window of 2.
2. **DCTCP**, which leverages ECN for enhanced performance in datacenter contexts.
3. **pFabric**, approximates shortest-job-first scheduling in a network context using switch support to prioritize flows with a smaller remaining flow size [16]. We set pFabric to have an initial congestion window of 12 packets and a retransmission timeout of 45μs.
4. **pHost**, emulates pFabric's behavior but using only scheduling at the end hosts [55] and hence allows the use of commodity switches. We set pHost to have a free token limit of 8 packets and a retransmission timeout of 9.5μs as recommended in [55].
5. **Fastpass**, introduces a centralized scheduler that schedules every packet. We implement Fastpass's [104] scheduling algorithm in our simulator as described in [55] and optimistically assume that the scheduler's decision logic itself incurs no overhead (i.e., takes zero time) and hence we only consider the latency and bandwidth overhead of contacting the central scheduler. We set the Fastpass epoch size to be 8 packets.

4.4.3 Network-level performance

We evaluate the performance of our candidate transport protocols in terms of their mean slowdown [16], which is computed as follows. The slowdown for a flow is computed by dividing the flow completion time achieved in simulation by the time that the flow would take to complete if it were alone in the network. The mean slowdown is then computed by averaging the slowdown over all flows. Figure 4.7 plots the mean slowdown for our five candidate protocols, using 100Gbps links (all other parameters are as in §4.4.2).

Results. We make the following observations. First, while the relative ordering in mean slowdown for the different protocols is consistent with prior results [55], their *absolute* values are higher than

reported in their original papers; e.g. pFabric and pHost both report close-to-optimal slowdowns with values close to 1.0 [16, 55]. On closer examination, we found that the higher slowdowns with disaggregation are a consequence of the differences in our traffic workloads (both earlier studies used heavy-tailed traffic workloads based on measurement studies from existing datacenters). In our DDC workload, reflecting the application-driven nature of our workload, we observe many flow arrivals that appear very close in time (only observable on sub-10s of microsecond timescales), leading to high slowdowns for these flows. This effect is strongest in the case of the Wordcount application, which is why it suffers the highest slowdowns. We observed similar results in our simulation of rack-scale disaggregation (graph omitted).

4.4.4 Application-level performance

We now use the pFabric FCTs obtained from the above simulations as the memory access times in our emulation methodology from §4.3.

We measure the degradation in application performance that results from injecting remote memory access times drawn from the FCTs that pFabric achieves with 40Gbps links and with 100Gbps links, in each case considering both datacenter-wide and rack-scale disaggregation. As in §4.3, we measure performance degradation compared to the baseline of performance without disaggregation (i.e., injecting zero latency).

In all cases, we find that the inclusion of queuing delay *does* have a non-trivial impact on performance degradation at 40 Gbps – typically increasing the performance degradation relative to the case of zero-queuing delay by between 2-3x, with an average performance degradation of 14% with datacenter-scale disaggregation and 11% with rack-scale disaggregation.

With 100Gbps links, we see (in Figure 4.8) that the performance degradation ranges between 1-8.5% on average with datacenter scale disaggregation, and containment to a rack lowers the degradation to between 0.4-3.5% on average. This leads us to conclude that 100Gbps links are both required and sufficient to contain the performance impact of queuing delay.

4.5 Future Directions

So far, we used emulation and simulation to evaluate the minimum network requirements for disaggregation. This opens two directions for future work: (1) demonstrating an end-to-end system implementation of remote memory access that meets our latency targets, and (2) investigating programming models that actively exploit disaggregation to *improve* performance. We present early results investigating the above with the intent of demonstrating the potential for realizing positive results to the above questions: each topic merits an in-depth exploration that is out of scope for this paper.

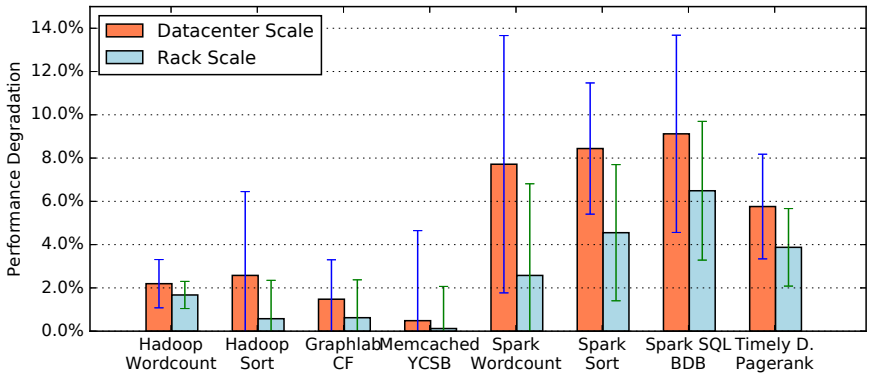


Figure 4.8: Application layer slowdown for each of the four applications at rack-scale and datacenter scale after injecting pFabric's FCT with 100Gbps link.

4.5.1 Implementing remote memory access

We previously identified an end-to-end latency target of 3-5μs for DDC that we argued could be met with RDMA. The (promising) RDMA latencies in §4.4 are as reported by native RDMA-based applications. We were curious about the feasibility of realizing these latencies if we were to retain our architecture from the previous section in which remote memory is accessed as a special swap device as this would provide a simple and transparent approach to utilizing remote memory.

We thus built a kernel space RDMA block device driver which serves as a swap device; i.e.,, the local CPU can now swap to remote memory instead of disk. We implemented the block device driver on a machine with a 3 GHz CPU and a Mellanox 4xFDR Infiniband card providing 56 Gbps bandwidth. We test the block device throughput using `dd` with direct IO, and measure the request latency by instrumenting the driver code. The end-to-end latency of our approach includes the RDMA request latency and the latency introduced by the kernel swap itself. We focus on each in turn.

RDMA request latency. A few optimizations were necessary to improve RDMA performance in our context. First, we *batch* block requests sent to the RDMA NIC and the driver waits for all the requests to return before notifying the upper layer: this gave a block device throughput of only 0.8GB/s and latency around 4-16us. Next, we *merge* requests with contiguous addresses into a single large request: this improved throughput to 2.6GB/s (a $3x$ improvement). Finally, we allow *asynchronous* RDMA requests: we created a data structure to keep track of outgoing requests and notify the upper layer immediately for each completed request; this improves throughput to 3.3GB/s which is as high as a local RamFS, and reduces the request latency to 3-4us (Table 4.5). This latency is within 2x of latencies reported by native RDMA applications which we view as encouraging given the simplicity of the design and that additional optimizations are likely possible.

Swap latency. We calculated the software overhead of swapping on a commodity desktop running Linux 3.13 by simultaneously measuring the times spent in the page fault handler and accessing

Min	Avg	Median	99.5 Pcntl	Max
3394	3492	3438	4549	12254

Table 4.5: RDMA block device request latency(ns)

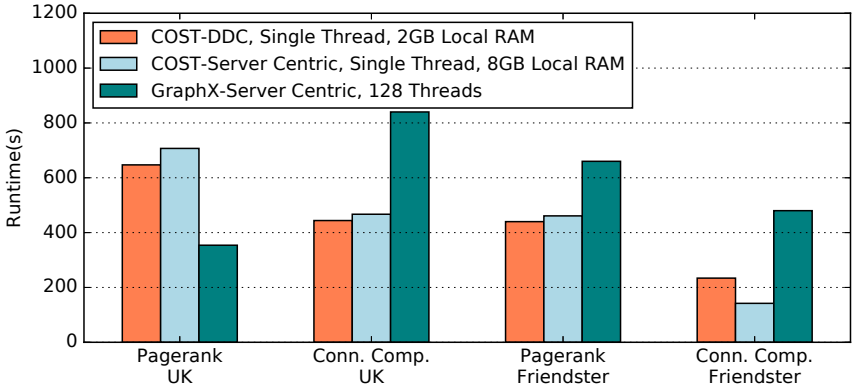


Figure 4.9: Running COST in a simulated DDC. COST-DDC is 1.48 to 2.05 faster than GraphX-Server Centric except for one case. We use two datasets in our evaluation, UK-2007-05 (105m nodes, 3.7b edges), and Friendster (65m nodes, 1.8b edges)

disk. We found that convenient measurement tools such as `ftrace` and `printk` introduce unacceptable overhead for our purposes. Thus, we wrap both the body of the `__do_page_fault` function and the call to the `swapin_readahead` function (which performs a swap from disk) in `ktime_get` calls. We then pack the result of the measurement for the `swapin_readahead` function into the unused upper 16-bits of the return value of its caller, `do_swap_page`, which propagates the value up to `__do_page_fault`.

Once we have measured the body of `__do_page_fault`, we record both the latency of the whole `__do_page_fault` routine (25.47μs), as well as the time spent in `swapin_readahead` (23.01μs). We subtract these and average to find that the software overhead of swapping is 2.46μs. This number is a lower-bound on the software overhead of the handler, because we assume that all of `swapin_readahead` is a "disk access".

In combination with the above RDMA latencies, these early numbers suggest that a simple system design for low-latency access to remote memory could be realized.

4.5.2 Improved performance via disaggregation

In the longer term, one might expect to re-architect applications to actively exploit disaggregation for improved performance. One promising direction is for applications to exploit the availability of low-latency access to large pools of remote memory [84]. One approach to doing so is based on extending the line of argument in the COST work [87] by using remote memory to avoid

parallelization overheads. COST is a single machine graph engine that outperforms distributed graph engines like GraphX when the graph fits into main memory. The RDMA swap device enables COST to use "infinite" remote memory when the graph is too large. We estimate the potential benefits of this approach with the following experiment. First, to model an application running in a DDC, we set up a virtual machine with 4 cores, 2GB of local memory, and access to an "infinitely" large remote memory pool by swapping to an RDMA-backed block device. Next, we consider two scenarios that represent server-centric architecture. One is a server with 4 cores and 8GB of local memory (25% larger than the DDC case as in previous sections) and an "infinitely" large local SSD swap – this represents the COST baseline in a server-centric context. Second, we evaluate GraphX using a 16-node `m2.4xlarge` cluster on EC2 – this represents the scale-out approach in current server-centric architecture. We run Pagerank and Connected Components using COST, a single-thread graph compute engine over three large graph datasets. COST `mmaps` the input file, so we store the input files on another RDMA-backed block device. Figure 4.9 shows the application runtime of COST-DDC, COST-SSD and GraphX-Server Centric. In all but one case, COST-DDC is 1.48 to 2.05 times faster than the GraphX (server-centric) scenario and slightly better than the server-centric COST scenario (the improvement over the latter grows with increasing data set size). Performance is worse for Pagerank on the UK-2007-5 dataset, consistent with the results in [87] because the graph in this case is more easily partitioned.

Finally, another promising direction for improving performance is through better resource utilization. As argued in [63, 84], CPU-to-memory utilization for tasks in today's datacenters varies by three orders of magnitude across tasks; by "bin packing" on a much larger scale, DDC should achieve more efficient statistical multiplexing, and hence higher resource utilization and improved job completion times. We leave an exploration of this direction to future work.

4.6 Related Work and Discussion

As mentioned earlier, there are many recent and ongoing efforts to prototype disaggregated hardware. We discussed the salient features of these efforts inline throughout this paper and hence we only briefly elaborate on them here.

Lim et al. [84, 85] discuss the trend of growing peak compute-to-memory ratio, warning of the "memory capacity wall" and prototype a disaggregated memory blade. Their results demonstrate that memory disaggregation is feasible and can even provide a 10x performance improvement in memory constrained environments.

Sudan et al. [123] use an ASIC based interconnect fabric to build a virtualized I/O system for better resource sharing. However, these interconnects are designed for their specific context; the authors neither discuss network support for disaggregation more broadly nor consider the possibility of leveraging known datacenter network technologies to enable disaggregation.

FireBox [19] proposes a holistic architecture redesign of datacenter racks to include 1Tbps silicon photonic links, high-radix switches, remote nonvolatile memory, and System-on-Chips (SoCs).

Theia [134] proposes a new network topology that interconnects SoCs at high density. Huawei's DC3.0 (NUWA) system uses a proprietary PCIe-based interconnect. R2C2 [44] proposes new topologies, routing and congestion control designs for rack-scale disaggregation. None of these efforts evaluate network requirements based on existing workloads as we do, nor do they evaluate the effectiveness of existing network designs in supporting disaggregation or the possibility of disaggregating at scale.

In an early position paper, Han et al. [63] measure – as we do – the impact of remote memory access latency on application-level performance within a single machine. Our work extends this understanding to a larger set of workloads and concludes with more stringent requirements on latency and bandwidth than Han et al. do, due to our consideration of Class B applications. In addition, we use simulation and emulation to study the impact of queueing delay and transport designs which further raises the bar on our target network performance.

Multiple recent efforts [48, 77, 83, 101] aim to reduce the latency in networked applications through techniques that bypass the kernel networking stack, and so forth. Similarly, efforts toward NIC integration by CPU architects [37] promise to enable even further latency-saving optimizations. As we note in §4.3.3, such efforts are crucial enablers in meeting our latency targets.

Distributed Shared Memory (DSM) [34,81,93] systems create a shared address space and allow remote memory to be accessed among different endpoints. While this is a simple programming abstraction, DSM incurs high synchronization overhead. Our work simplifies the design by using remote memory only for paging, which removes synchronization between the endpoints.

Based on our knowledge of existing designs and prototypes [19, 67, 84, 85, 126], we assume partial memory disaggregation and limit the cache coherence domain to one CPU. However, future designs may relax these assumptions, causing more remote memory access traffic and cache coherence traffic. In these designs, specialized network hardware may become necessary.

4.7 Conclusion

We did a preliminary study that identifies numerous directions for future work before disaggregation is deployable. Most important among these are the adoption of low-latency network software and hardware at endpoints, the design and implementation of a "disaggregation-aware" scheduler, and the creation of new programming models which exploit a disaggregated architecture. We believe that quantified, workload-driven studies such as that presented in this paper can serve to inform these ongoing and future efforts to build DDC systems.

Chapter 5

pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric

5.1 Introduction

Users of Internet services are extremely sensitive to delays. Motivated by this, there has been a tremendous effort recently to optimize network performance in modern datacenters. Reflecting the needs of datacenter applications, these efforts typically focus on optimizing the more application-centric notion of flow completion time (FCT), using metrics such as a flow's *slowdown* which compares its FCT against the theoretical minimum (flow size in bytes divided by the access link bandwidth).

Recent research has produced a plethora of new datacenter transport designs [15, 16, 27, 65, 91, 104, 137]. The state-of-the-art is pFabric [16] that achieves close to theoretically minimal slowdown over a wide variety of workloads. However, to achieve this near-optimal performance, pFabric requires specialized network hardware that implements a specific packet scheduling and queue management algorithm[1]. There are two disadvantages to this: (i) pFabric cannot use commodity hardware, and (ii) pFabric's packet scheduling algorithm cannot be altered to achieve policy goals beyond minimizing slowdown — such goals may become relevant when the datacenter is shared by multiple users and/or multiple applications.

Countering this use of specialized hardware, the Fastpass proposal [104] uses commodity switches coupled with a flexible and fine-grained (close to per-packet) central scheduler. While

[1]Specifically, in pFabric, each packet carries the number of currently remaining (that is, un-ACKed) bytes in the packet's flow. A pFabric switch defines a flow's priority based on the packet from that flow with the smallest remaining number of bytes. The switch then schedules the oldest packet from the flow with the highest priority. Note that, within a flow, the oldest packet may be different from the packet with the fewest remaining bytes since packets transmitted later may record fewer remaining bytes.

this allows the network fabric to remain simple and policy-agnostic, the resulting performance is significantly worse than that achieved by pFabric, especially for short flows (§5.4). In this paper, we ask: *Is it possible to achieve the near-ideal performance of pFabric using commodity switches?*

We answer this question in the affirmative with pHost, a new datacenter transport design that is simple and general — requiring no specialized network hardware, no per-flow state or complex rate calculations at switches, no centralized global scheduler and no explicit network feedback — yet achieves performance surprisingly close to pFabric.

In the next section, we provide the rationale for and overview of the pHost design. We provide the specifics of our design in §5.3. In §5.4 we evaluate pHost's performance, comparing it against pFabric [16] and Fastpass [104]. We discuss related work in §5.5 and close the paper with a few concluding comments in §5.6.

5.2 pHost Overview

We start with a brief review of modern datacenter networks (§5.2.1). We then describe the key aspects of pHost's design (§5.2.2), and close the section with an intuitive description of why pHost's approach works (§5.2.3).

5.2.1 Modern Datacenter Networks

Modern datacenter networks differ from traditional WAN networks in several respects.

- *Small RTTs*: The geographic extent of datacenter networks is quite limited, so the resulting speed-of-light latencies are small. In addition, switch forwarding latencies have dropped as cut-through switching has become common in commodity switches.
- *Full bisection bandwidth*: By using topologies such as Fat-Tree [13] or VL2 [62], datacenter networks now provide full bisection bandwidth [11, 107].
- *Simple switches*: Datacenter switches tend to be relatively simple (compared to high-end WAN routers). However, they do provide some basic features: a few priority levels (typically 8–10 [5,15,65]), ECMP and/or packet spraying (that is, randomized load balancing on a per-flow and/or per-packet basis [5, 46]), cut-through switching, and relatively small buffers.

pHost both assumes and exploits these characteristics of datacenter networks, as we elaborate on in §5.2.3.

5.2.2 Basic Transport Mechanism

We now provide a high-level overview of pHost's transport mechanism. pHost is built around a host-based scheduling mechanism that involves requests-to-send (RTS), per-packet token assign-

ment, and receiver-based selection of pending flows. These techniques have their roots in wireless protocols (e.g., 802.11) and capability-based DDoS mechanisms (e.g., SIFF [141]). Specifically:

- Each source end-host, upon a flow arrival, sends a request to send (RTS) packet to the destination of the flow. The RTS may contain information relevant for making scheduling decisions (such as flow's size, or which tenant the source belongs to, etc.).
- Once every packet transmission time, each destination end-host considers the set of pending RTSs (that is, RTSs for flows that still have bytes to send) and sends a "token" to one of the corresponding sources. The token allows the source to transmit one data packet from that flow, and may specify the priority level at which the packet is to be sent at. Thus, each destination host performs *per-packet* scheduling across the set of active flows *independent* of other destinations.
- Tokens expire if the source has not used the token within a short period after receiving the token (default being 1.5× MTU-sized packet transmission time). Each source may also be assigned a few "free tokens" for each flow, which need not be sent from the destination host.
- After each packet transmission, a source selects one of its unexpired tokens and sends the corresponding data packet. Thus, each source host also performs a selection across its set of active flows *independent* of our sources.
- Once the destination has received all data packets for a flow, it sends an ACK packet to the source.
- All control packets (RTS, tokens, ACKs) are sent at the highest priority.

pHost's design has several degrees of freedom: the scheduling at the destination (which flows to send next token to), the scheduling at the source (which token to use after each packet transmission), the priority level at which each data packet is sent at, and the number of free tokens assigned to the sources. These can be configured to achieve different performance goals, without any modification in the network fabric. For instance, we demonstrate later that pHost is competitive with pFabric when the above degrees of freedom are configured to globally minimize slowdown, but can also be configured to optimize for performance metrics other than slowdown (*e.g.*, meeting flow deadlines, or achieving fairness across multiple tenants, etc.).

5.2.3 Why pHost works

Similar to prior proposals (e.g., pFabric [16]), we utilize the packet-spraying feature found in many commodity switches (in which packets are spread uniformly across the set of available routes) [5, 46]. Intuitively, using packet-spraying in a full-bisection-bandwidth network can eliminate almost all congestion in the core (§5.4), so we do not need sophisticated path-level scheduling (as in Fastpass) nor detailed packet scheduling in the core switches (as in pFabric). However, we do make use of the few levels of priority available in commodity switches to ensure that signaling packets suffer few drops.

While there is no congestion in the core, there can still be congestion at the destination host if multiple sources are sending flows to the destination at the same time. In pHost, this comes down to how the destination grants tokens in response to RTS requests from sources. Choosing flows that should be assigned tokens at any time is effectively a bipartite matching problem. A centralized scheduler could compute a match based on a global view (akin to how early routers managed their switch fabrics) but this would incur the complexity of a scalable centralized scheduler and the latency overhead of communication with that scheduler (§5.4). In pHost, we instead use a fully decentralized scheduler (once again, akin to router scheduler designs such as PIM [17] and iSlip). The resulting match may be imperfect, but we compensate for this in two ways. To avoid starvation at the source (if a destination does not respond with a token), we allow the source to launch multiple RTSs in parallel. Each source is also given a small budget of free tokens for each flow; this also allows sources to start sending without waiting for the RTT to hear from the destination. To avoid starvation at the destination (e.g., when a source does not utilize the token it was assigned), we use a back-off mechanism where (for a short time) a destination avoids sending tokens to a particular source if the source has not used the tokens it was recently assigned by the destination.

As we shall show, the combination of these techniques avoids starvation at the hosts, and allow pHost to achieve good network utilization despite a fully decentralized host-based scheduler.

5.3 Design Details

We now describe the details of pHost's design. At a high level, there are two main components to pHost's design: (i) the protocol that dictates *how* sources and destinations communicate by exchanging and using RTS, token and data packets, and (ii) the scheduling policy that dictates *which* sources and destinations communicate.

We start by describing the protocol that end-hosts implement (§5.3.1) and then elaborate on how this protocol ensures high network utilization (§5.3.2). We then describe how pHost supports flexible scheduling policies (§5.3.3). Finally we describe how pHost achieves reliable transmission in the face of packet drops (§5.3.4).

5.3.1 pHost Source and Destination

pHost end-hosts run simple algorithms for token assignment and utilization. The algorithm for the source is summarized in Algorithm 1. When a new flow arrives, the source immediately sends an RTS to the destination of the flow. The RTS may include information regarding the flow (flow size, deadline, which tenant the flow belongs to, etc) to be used by the destination in making scheduling decisions. The source maintains a per-flow list of "tokens" where each token represents the permission to send one packet to the flow's destination; we refer to this as the *ActiveTokens* list. The ActiveTokens list is initialized with a configurable number of "free tokens" (we elaborate on the role of free tokens in §5.3.2); all subsequent tokens can only be explicitly granted by the destination in response to an RTS.

Algorithm 1 pHost algorithm at Source.

```
if new flow arrives then
    Send RTS
    ActiveTokens ← FreeTokens                    ▷ Add free tokens (§5.3.2)
else if new token T received then
    Set ExpiryTime(T)                            ▷ Tokens expire in fixed time (§5.3.2)
    ActiveTokens ← T
else if idle then
    T = Pick(ActiveTokens)                       ▷ pick unexpired token (§5.3.3)
    Send Packet corresponding to T
end if
```

When a source receives a token from the destination, it adds this token to its ActiveTokens list. Each token has an associated expiry time and the source is only allowed to send a packet if it holds an unexpired token for that packet (again, we elaborate on the role of token timeouts in §5.3.2). Whenever a source is idle, it selects a token from ActiveTokens based on the desired policy goals (§5.3.3) and sends out the packet for the corresponding token.

The high-level algorithm used at the destination is summarized in Algorithm 2. Each destination maintains the set of flows for which the destination received an RTS but has not yet received all the data packets; we refer to this as the *PendingRTS* list. When the destination receives a new RTS, it adds the RTS to PendingRTS immediately. Every (MTU-sized) packet transmission time, the destination selects an RTS from PendingRTS list based on the desired policy goals (§5.3.3) and sends out a token to the corresponding source. The token contains the flow ID, the packet ID and (optionally) a priority value to be used for the packet. As at the source, each token has an associated expiry time (we use a value of $1.5 \times$ MTU-sized packet transmission time). A token assigned by the destination is considered revoked if it is not utilized within its expiry time. This avoids congestion at the destination as sources cannot use tokens at arbitrary times. In addition, the destination maintains a count of the number of expired tokens for the flow . If this count exceeds a threshold value, the flow is marked as "downgraded" which lowers the likelihood it will be granted tokens in the immediate future; we elaborate on the details and purpose of downgrading in §5.3.2. Finally, once the destination has received all the packets for a flow, it sends an ACK to the source and removes the RTS from PendingRTS list.

Algorithm 2 pHost algorithm at Destination.

```
if receive RTS R then
    PendingRTS ← R
else if idle then
    F = Pick(PendingRTS)                          ▷ Pick an active flow (§5.3.3)
    Send Token T for flow F
    if F.#ExpiredTokens > Threshold then          ▷ (§5.3.2)
        Downgrade F for time t*                   ▷ (§5.3.2)
    end if
else if Received data for token T then
    Set Token T as responded
end if
```

Note that all control packets (RTS, token, ACK) in pHost are sent at the highest priority.

5.3.2 Maximizing Network Utilization

As discussed earlier, packet spraying in a full-bisection bandwidth network eliminates almost all congestion in the core. However, sources sending multiple RTSs in parallel (Algorithm 1) and destinations assigning one token per packet transmission time (Algorithm 2) may lead to network underutilization. For a centralized scheduler (as in Fastpass), avoiding this underutilization is easy since the scheduler has a global view of the network. To achieve high network utilization in a fully decentralized manner, however, pHost has to resolve two challenges. We discuss these challenges and how pHost resolves these challenges.

Free tokens for new flow arrivals. Recall that upon a flow arrival, the source immediately sends an RTS to the corresponding destination. The first challenge is that the bandwidth at the source may be wasted until the token for the flow has been received (even if the destination is free, source cannot send the packets). This may have particularly adverse effect on short flow performance, since such a wait may be unnecessary. pHost avoids this overhead by assigning each source a few "free tokens" per flow that can be used by the source without receiving any tokens from the corresponding destination.

Source downgrading and token expiry. To understand the second challenge, let us consider the case of Figure 5.1. When the source in the above example prefers the token for flow B in the second time unit, the bandwidth at the destination for flow A is wasted. Even worse, another source may have a flow C to send to the destination of flow A, but the destination continues sending tokens to the source of flow A, which in turn continues to prefer utilizing tokens for flow B. This may lead to long term wastage of bandwidth at both the destination of flow A and the source of flow C.

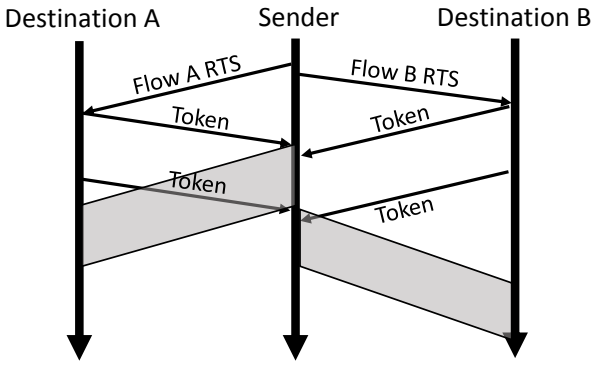


Figure 5.1: In this example, two flows A and B arrive at the source at roughly the same time, and two RTS are sent to the respective destinations. Suppose the source gets a token for flow A first. Since the source has only one token, it immediately consumes it by sending the corresponding data packet to destination A. Now suppose the source receives another token for flow A and a token for flow B while it is sending the data packet for flow A. The source now has two tokens, one for each flow. Suppose the source decides to utilize the token for flow B at this step.

pHost uses a source downgrading mechanism to prevent a destination from sending tokens to a source that does not respond with data packets. As mentioned earlier, pHost destinations maintain a count of the number of unexpired tokens for each source. If this count exceeds a threshold value in succession (default being a BDP worth of tokens), the destination *downgrades* the source and stops assigning tokens to that source. The source is left downgraded for a *timeout* period (default being $3\times$ RTT). After the timeout period, the destination resends tokens to the source for the packets that were not received.

5.3.3 Local Scheduling Problem

Datacenter network operators today have to contend between a rich mix of tenants and applications sharing the network, each of which may have a different performance goal. For instance, the goal in some scenarios may be to optimize for tail latency (*e.g.*, web search and social networking) across all flows. In other scenarios (*e.g.*, multiple tenants), the goal may be to share network bandwidth fairly among the tenants. As pHost implements scheduling at end-hosts, it naturally provides algorithmic flexibility in optimizing for various performance metrics. In this subsection, we describe how this enables flexibility in terms of network resource allocation between users and applications, and to optimize the network for a wide variety of performance metrics.

Recall, from §5.3.1, that the pHost sources send a RTS to the destination expressing their intent of sending packets upon each flow arrival. The sources embed the information related to the flow (*e.g.*, flow size, deadlines, etc) within the RTS packet. The destinations then assign tokens to the flows, optionally specifying a priority level to be used for the packets in the flow. We

describe how this design enables optimizing for three different performance objectives using end-host scheduling: (i) minimizing flow completion time [15,16,27,91,104]; (ii) deadline-constrained traffic [16, 65, 137]; and (iii) fairness across multiple tenants.

The optimal algorithm for minimizing flow completion time when scheduling over a single link is Shortest Remaining Processing Time (SRPT) scheduling, which prioritizes the flow with the fewest remaining bytes. Transport protocols [16, 104] that achieve near-ideal performance emulate this policy over a distributed network fabric by prioritizing flows that have least number of packets remaining to complete the flow. pHost can emulate SRPT over a distributed network fabric using the same scheduling policy — each destination prioritizes flows with least number of remaining packets while assigning tokens; the destination additionally allows the sources to send short flows with the second highest priority and long flows with the third highest priority (recall, control packets are sent with the highest priority). Note that this is significantly different from pFabric, which assigns packet priority to be the remaining flow size. Similarly, the sources prioritize flows with the fewest number of remaining packets while utilizing tokens; the sources also use any free tokens when idle. We show in §5.4 that using this simple scheduling policy at the end-hosts, pHost achieves performance close to that of state-of-the-art protocols when minimizing flow completion time.

Next, we discuss how pHost enables optimizing for deadline-constrained traffic [16, 65, 137]. The optimal algorithm for scheduling deadline-constrained flows over a single link is Earliest Deadline First (EDF), which prioritizes the flow with the earliest deadline. pHost can emulate EDF by having each source specify the flow deadline in its RTS. Each destination then prioritizes flows with earliest deadline (analogous to the SRPT policy) when assigning tokens; the sources, as earlier, prioritize flows with earliest deadline when utilizing tokens.

Indeed, pFabric [16] can emulate the above two policies despite embedding the scheduling policies within the network fabric. We now discuss a third policy which highlights the necessity of decoupling scheduling from the network fabric. Consider a multi-tenant datacenter network where tenant A is running a web search workload (most flows are short) while tenant B is running a MapReduce workload (most flows are long). pFabric will naturally prioritize tenant A's shorter flows over tenant B's longer flows, starving tenant B. pHost, on the other hand, can avoid this starvation using its end-host based scheduling. Specifically, the destinations now maintain a counter for the number of packets received so far from each tenant and in each unit time assign a token to a flow from the tenant with smaller count. While providing fairness across tenants, pHost can still allow achieving the tenant-specific performance goals for their respective flows (implementing scheduling policies for each tenant's flows).

5.3.4 Handling Packet drops

As we show in §5.4, the core idea of pHost end-hosts performing per-packet scheduling to minimize congestion at their respective access links leads to negligible number of packet drops in full-bisection bandwidth datacenter networks. For the unlikely scenario of some packets being dropped, per-packet token assignment in pHost lends itself to an extremely simple mechanism to

handle packet drops. In particular, recall from §5.3.1, that each destination in pHost assigns a token *to a specific packet* identifying the packet ID along with the token. If the destination does not receive one of the packets until a token has been sent out for the last packet of the flow (or timeout), the destination simply reissues a token for the lost packet when the flow has the turn to receive a token. The source, upon receiving the token, retransmits the lost packet(s).

5.4 Evaluation

In this section, we evaluate pHost over a wide range of datacenter network workloads and performance metrics, and compare its performance against pFabric and Fastpass.

5.4.1 Test Setup

Our overall test setup is identical to that used in pFabric [16][2]. We first elaborate on this setup — the network topology, workload and metrics; we then describe the protocols we evaluate and the default test configurations we use.

Network Topology. We use the same network topology as in pFabric [16]. The topology is a two-tier multi-rooted tree with 9 racks and 144 end-hosts. Each end-host has a 10Gbps access link and each core switch has nine 40Gbps links; each network link has a propagation delay of 200ns. The resultant network fabric provides a full bisection bandwidth of 144Gbps. Network switches implement cut-through routing and packet spraying (functions common in existing commodity switches [5, 16, 46]). pFabric assumes each switch port has a queue buffer of 36kB; we use this as our default configuration value but also evaluate the effect of per-port buffer sizes ranging from 6kB-72kB.

Workloads. We evaluate performance over three production traces whose flow size distributions are shown in Figure 5.2. All three traces are heavy-tailed, meaning that most of the flows are short but most of the bytes are in the long flows. The first two – "Web Search" [15] and "Data Mining" [62]) – are the traces used in pFabric's evaluation. The third "IMC10" trace uses the flow size distributions reported in a measurement study of production datacenters [35]. The IMC10 trace is similar to the Data Mining trace except in the tail: the largest flow in the IMC10 trace is 3MB compared to 1GB in the Data Mining trace. In addition to production traces, we also consider a synthetic "bimodal" workload that we use to highlight specific performance characteristics of the three protocols; we elaborate on this workload inline in §5.4.3. As in prior work [16, 27], we generate flows from these workloads using a Poisson arrival process for a specified target network load. We consider target network loads ranging from $0.5 - 0.8$.

Performance metrics. The primary metric we focus on is that used in pFabric: mean *slowdown*, defined as follows: let `OPT(i)` be the flow completion time of flow `i` when it is the only flow

[2]We would like to thank the pFabric authors for sharing their simulator with us; our simulator (`https://github.com/NetSys/simulator`) builds upon theirs.

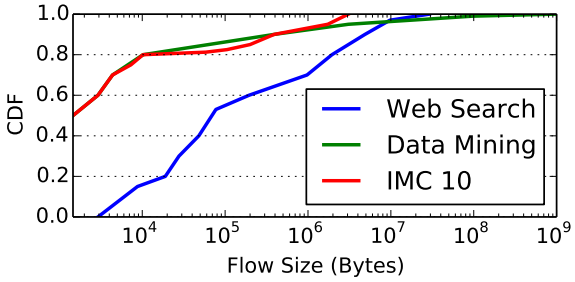


Figure 5.2: Distribution of flow sizes across workloads used in our evaluation. Note that short flows dominate all workloads; however, Data Mining and IMC10 workloads have significantly larger fraction of short flows when compared to the Web Search workload.

in the network and let `FCT(i)` be the observed flow completion time when competing with other flows. Then, for flow `i`, the *slowdown* is defined as the ratio of `FCT(i)` and `OPT(i)`. Note that `FCT(i)` $\geq$ `OPT(i)`; thus, a smaller slowdown implies better performance. The mean and high percentile slowdowns are calculated accordingly across the set of flows in the workload. For completeness, in §5.4.3, we consider a range of additional metrics considered in previous studies, including "normalized" FCT, throughput, and the fraction of flows that meet their target deadlines; we define these additional metrics in §5.4.3.

Evaluated Protocols. We evaluate pHost against pFabric [16] and Fastpass [104]. For pFabric, we use the simulator provided by the authors of pFabric with their recommended configuration options: an initial congestion window of 12 packets, an RTO of 45μs, and the network topology described above. Unfortunately, a packet-level simulator is unavailable for Fastpass and hence we implemented Fastpass in our own simulator. Our implementation of Fastpass uses: (1) 40B control packets and an epoch size of 8 packets (Fastpass makes scheduling decisions every epoch with a recommended epoch interval of 8 MTU transmission times), (2) zero processing overhead at the centralized packet scheduler (i.e., we assume the scheduler solves the global scheduling problem infinitely fast); and (3) perfect time synchronization (so that all end-hosts are synchronized on epoch start and end times). Note that the latter two represent the *best-case* performance scenario for Fastpass.

Default configuration. Unless stated otherwise, our evaluation use a default configuration that is based on an all-to-all traffic matrix with a network load of 0.6, a per-port buffer of 36kB at switches, and other settings as discussed above. For pHost, we set the token expiry time to be 1.5$\times$, source downgrade time to be 8$\times$ and timeout to be 24$\times$ MTU-sized packet transmission time (note that BDP for our topology is 8 packets). Moreover, we assign 8 free tokens to each flow. We evaluate the robustness of our results over a range of performance metrics, workloads, traffic matrices and parameter settings in §5.4.3.

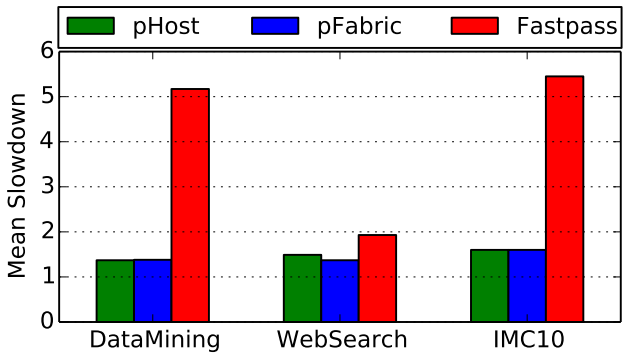
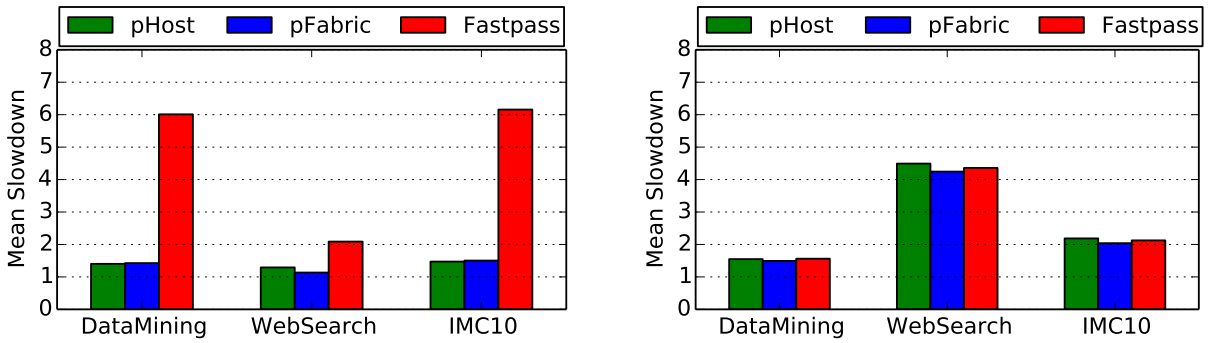


Figure 5.3: Mean slowdown of pFabric, pHost, and Fastpass across different workloads for our default configuration (0.6 load, per-port buffers of 36kB). pHost performs comparable to pFabric, and 1.3–4× better than Fastpass.



(a) Short flows: Mean Slowdown

(b) Long flows: Mean Slowdown

Figure 5.4: Breakdown of mean slowdown by flow size for pFabric, pHost, and Fastpass (all flows greater than 10MB (for Data Mining and Web Search workloads) and greater than 100KB (for IMC10 workload) are considered long flows). All the three schemes have similar performance for long flows; for short flows, however, pHost performs similar to pFabric and 1.3–4× better than Fastpass.

5.4.2 Performance Evaluation Overview

Figure 5.3 shows the mean slowdown achieved by each scheme for our three trace-based workloads. We see that the performance of pHost is comparable to that of pFabric. pFabric is known to achieve near-optimal slowdown [16]; hence these results show that pHost's radically different design approach based on scheduling at the end-hosts is equally effective at optimizing slowdown.

Somewhat surprisingly, we see that slowdown with Fastpass is almost $4\times$ higher than pHost and pFabric.[3] We can explain this performance difference by breaking down our results by flow size: Figure 5.4 shows the mean slowdown for short flows versus that for long flows. For long flows, all the three protocols have comparable performance; however, for short flows, both pHost and pFabric achieve significantly better performance than Fastpass. Since the three workloads contain approximately 82% short flows and 18% long flows, the performance advantage that pFabric and pHost enjoy for short flows dominates the overall mean slowdown.

That pFabric and pHost outperform Fastpass for short flows is (in retrospect) not surprising: Fastpass schedules flows in epochs of 8 packets, so a short flow must wait for at least an epoch ($\sim 10\mu$s) before it gets scheduled. Further, the signaling overhead of control packets adds another round trip of delay before a short flow can send any packet. Neither pFabric nor pHost incur this overhead on flow arrival. That all three protocols have comparable performance for long flows is also intuitive because, for a long flow, the initial waiting time in Fastpass (one epoch and one round trip time) is negligible compared to its total FCT.

The above results show that pHost can match the near-optimal performance of pFabric (without requiring specialized support from the network fabric) and significantly outperforms Fastpass (despite lacking the global view in scheduling packets that Fastpass enjoys). Next, we evaluate whether the above conclusions hold for a wider range of workloads, performance metrics and traffic matrices.

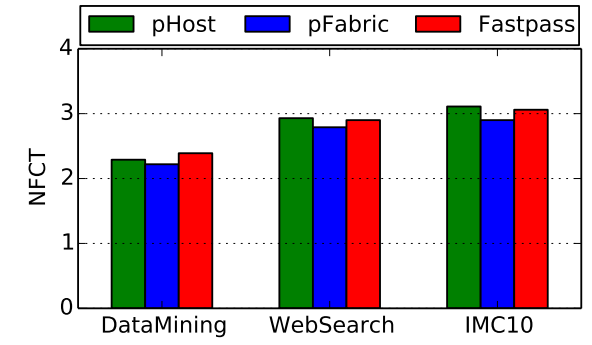
5.4.3 Varying Metrics and Scenarios

We now evaluate pHost– and how pHost compares to pFabric and Fastpass – over varying performance metrics, network load, traffic matrices, etc.

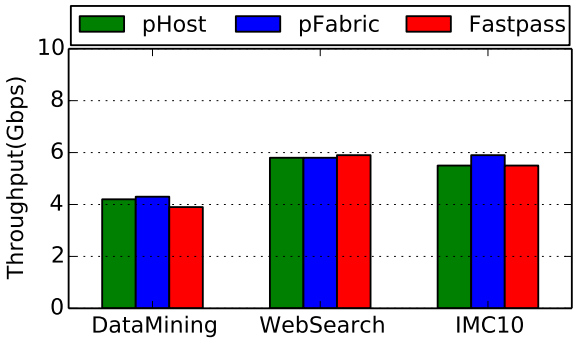
Varying Performance Metrics. Our evaluation so far has focused on mean slowdown as our performance metric. We now evaluate performance using five additional metrics introduced in prior work: (i) normalized flow completion time [27, 65, 91, 104], defined as ratio of the mean of `FCT(i)` and the mean of `OPT(i)`; (ii) network throughput, measured as the number of bytes delivered to receivers through the network over unit time normalized by the access link bandwidth; (iii) the 99 percentile in slowdown [16]; (iv) for deadline-constrained traffic, the fraction of flows that meet deadlines [16, 65, 137]; and (v) packet drop rates. Figure 5.5 shows our results using the above metrics.

[3] Note that the evaluation in [104] does not compare the performance of Fastpass to that of pFabric.

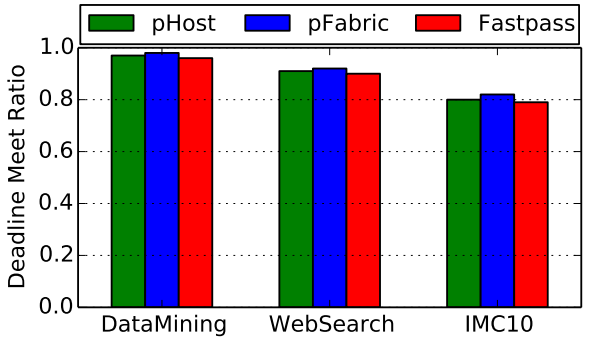
NFCT. Figure 5.5a shows that all three protocols see similar performance as measured by NFCT; across all evaluated cases, the maximum difference in NFCT between any two protocols is 15%. This similarity is simply because the NFCT metric, as defined, is (unlike mean slowdown) dominated by the FCT of long flows. FCT for long flows is in turn dominated by the time to transmit the large number of bytes involved, which is largely unaffected by protocol differences and hence all three protocols have similar performance.



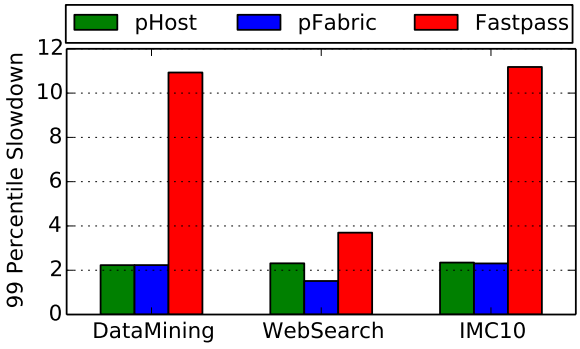
(a) NFCT



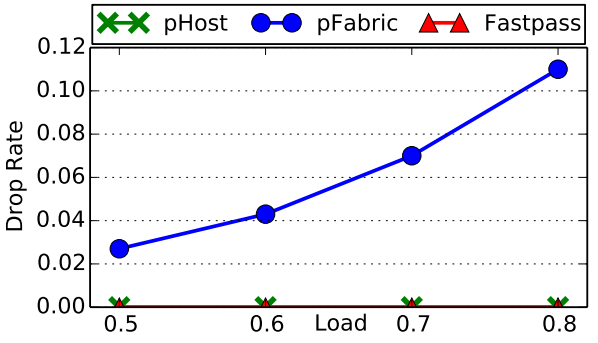
(b) Throughput



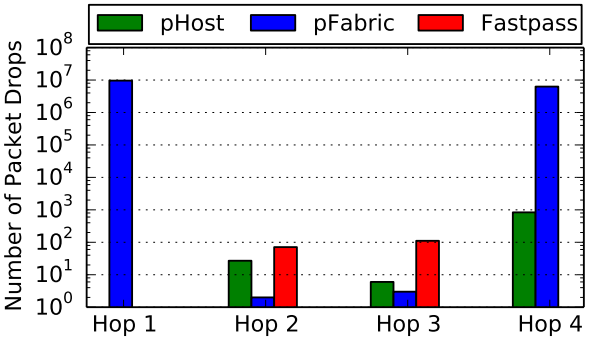
(c) Fraction of flows meeting deadline



(d) 99%-ile slowdown for short flows



(e) Drop rate



(f) Packet drops across hops

Figure 5.5: Performance of the three protocols across various performance metrics. See §5.4.3 for detailed discussion.

Throughput. The results for throughput (shown in Figure 5.5b) follow trend similar to NFCT results, again because overall throughput is dominated by the performance of long flows.

Deadlines. We now evaluate the performance of the three protocols over deadline-constrained traffic (Figure 5.5c). We assign a deadline to each flow using exponential distribution with mean 1000μs [16]; if the assigned deadline is less than $1.25\times$ the optimal FCT to a flow, we set the deadline for that flow to be $1.25\times$ its optimal FCT. We observe that all protocols achieve similar performance in terms of fraction of flows that meet their deadlines, with the maximum difference in performance between any two protocols being 2%.

We conclude that for applications that care about optimizing NFCT, throughput or deadline-constrained traffic, all three protocols offer comparable performance. The advantage of pHost for such applications lie in considerations other than performance: that pHost relies only on commodity network fabrics and that pHost avoids the engineering challenges associated with scaling a centralized controller.

Tail latency and drop rates. We now evaluate the three protocols for two additional performance metrics: tail latency for short flows and the packet drop rate.

99%ile Slowdown. Prior work has argued the importance of tail performance in datacenters and hence we also look at slowdown at the 99-percentile, shown in Figure 5.5d. We see that for both pHost and pFabric, the 99%ile slowdown is around 2 (roughly 33% higher than the mean slowdown), while for Fastpass the slowdown increases to almost $2\times$ the mean slowdown.

Drop rate. We now measure the drop rates for the three protocols. By design, one would expect to see very different behavior in terms of packet drops between pFabric and the other two protocols — pHost and Fastpass. Indeed, pFabric is deliberately aggressive in sending packets, expecting the network to drop low priority packets in large numbers; in contrast, pHost and Fastpass explicitly schedule packets to avoid drops. Figure 5.5e shows the overall drop rate of each protocol under increasing network load for the Web Search workload. As expected, we see that pFabric has a high drop rate that increases with load while pHost and Fastpass see drop rates consistently close to zero even as load increases.

Figure 5.5f shows *where* drops occur in the network: we plot the absolute number of packet drops at each of the 4 hops in the network (end-host NIC queue, the aggregation switch upstream queue, the core switch queue, and the aggregation switch downstream queue); 511 million packets are injected into the network over the duration of the simulation (network load being 0.6). We see that for pFabric, the vast majority of packet drops occur in the first (61%) and last (39%) hop queue, with almost no drops in the two intermediate hops. In contrast, because pHost and Fastpass explicitly schedule packets, first and last hop drops are almost eliminated: both protocols experience zero drops at the first hop, the number of last hop drops for pHost and Fastpass are 836 and 0 packets respectively.

Finally, we note that the absolute number of drops within the network fabric is low for all three protocols: 33, 5 and 182 drops for pHost, pFabric, and Fastpass respectively, which represent less

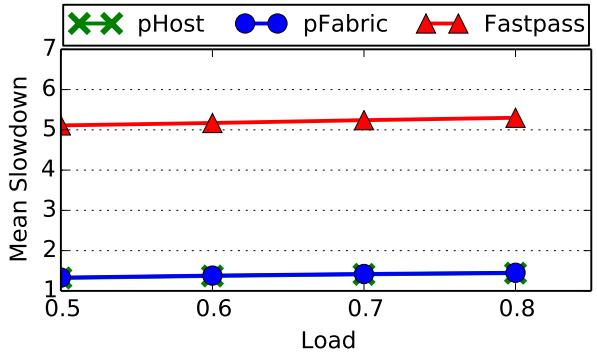
than 0.00004% of the total packets injected into the network. This confirms our intuition that full bisection bandwidth networks together with packet spraying avoids most congestion (and hence the need for careful scheduling) within the network fabric.

Varying network load. Our evaluation so far used traffic generated at 0.6 network load. We now evaluate protocol performance for network load varying from 0.5–0.9, as reported in prior work. Figure 5.6 presents our results across the three workloads and protocols.

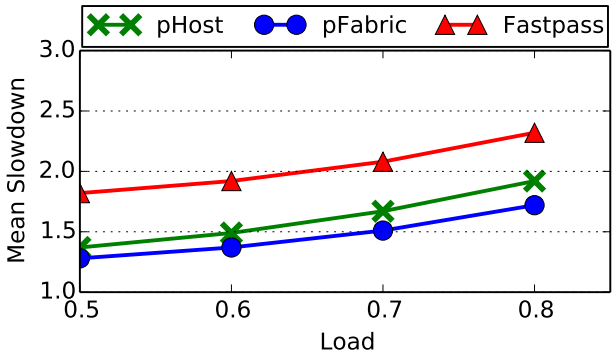
We observe that the relative performance of the different protocols across different network loads remains consistent with our results from above. This is to be expected as the distribution of short versus long flows remains unchanged with varying load.

We also note that, in all cases, performance degrades as network load increases. Closer examination revealed that in fact the overall network becomes *unstable* at higher loads; that is, with the network operating in a regime where it can no longer keep up with the generated input load. In Figure 5.7, we aim to capture this effect. In particular, the x-axis plots the fraction of packets (out of the total number of packets over the simulation time) that have arrived at the source as the simulation progresses; the y-axis plots the fraction of packets (again, out of the total number of packets across the simulation time) that have arrived at the source but have not yet been injected into the network by the source ("pending" packets). In a stable network, the fraction of pending packets would remain roughly constant over the duration of the simulation showing that the sources inject packets into the network at approximately the same rate at which they arrive. We observe that, at 0.6 load, this number does in fact remain roughly constant over time. However, at higher load, this number increases as the simulation progresses, indicating that packets arrive faster than the rate at which sources can inject then into the network. Measuring slowdown when operating in this unstable regime is unwise since the measured value depends closely on the duration of the simulation (e.g., in our experiments at 0.8 load, we could obtain arbitrarily high slowdowns for pFabric by simply tuning the length of the simulation run). This is the reason we select a network load of 0.6 as our default configuration (compared to the 0.8 used in pFabric).

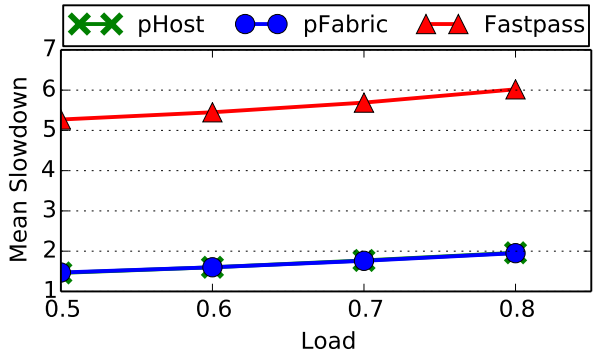
New Workloads. Our results so far were based on three traces used in prior work. While these reflect existing production systems, network workloads will change as applications evolve and hence we sought to understand how pHost's performance will change for workloads in which the ratio of short *vs.* long is radically different from that observed today. We thus created a synthetic trace that uses a simple bimodal distribution with short (3 packet) flows and long (700 packet) flows and vary the fraction of short flows from 0% to 99.5%. We show the corresponding mean slowdown in Figure 5.8. We make two observations. The first is that pHost once again matches pFabric's performance over the spectrum of test cases. The second — and perhaps more interesting — observation is that the absolute value of slowdown (for all protocols) varies significantly as the distribution of short vs. long flows changes; for pFabric and pHost, the traces based on current workloads occupy the "sweet spot" in the trend. This shows that although pFabric and our own pHost achieve near-optimal performance (i.e., mean slowdown values close to 1.0) for existing traces, this is not the case for radically different workloads. Whether and how one might achieve



(a) Data Mining



(b) Web Search



(c) IMC10

Figure 5.6: Performance of the three protocols across varying network loads.

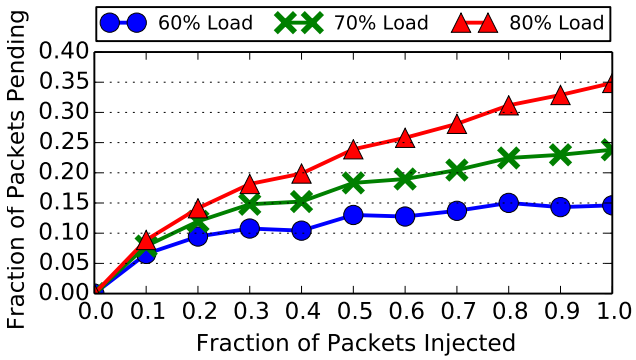


Figure 5.7: Stability analysis for pFabric. x-axis is the fraction of packets (out of total number of packets across the simulation) that have arrived at the source as the simulation progresses; y-axis is the fraction of packets (again, out of total number of packets across the simulation) that have yet not been injected into the network by the sources. pFabric is stable at 0.6 load, unstable beyond 0.7 load. We get similar results for pHost and Fastpass.

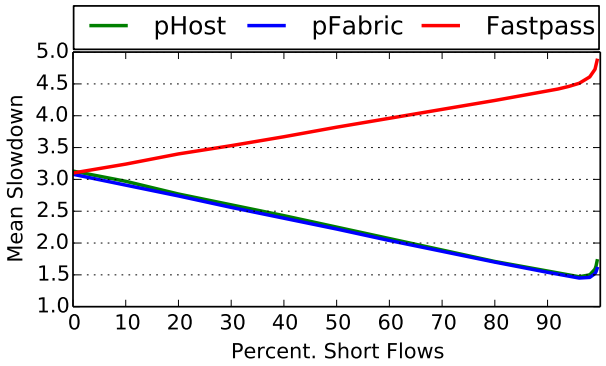
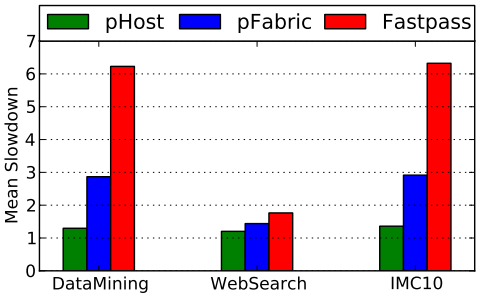


Figure 5.8: Mean slowdown of pHost, pFabric, and Fastpass in synthetic workload (with varying fraction of short flows). Both pFabric and pHost perform well when the trace is short flow dominated. Fastpass performs similar to pHost and pFabric when there are 90% long flows, but gets significantly worse as the fraction of short flows increases.

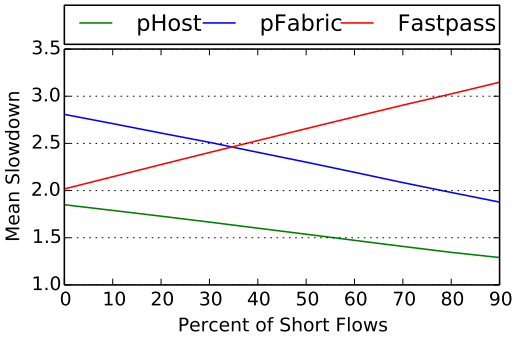
better performance for such workloads remains an open question for future work.

Varying switch parameters. We evaluate the impact of varying the per-port buffer size in switches. Figure 5.10 shows the mean slowdown with increasing switch buffer sizes for our Data Mining workload.[4] We see that none of the three schemes is sensitive to the sizing of switch buffers,

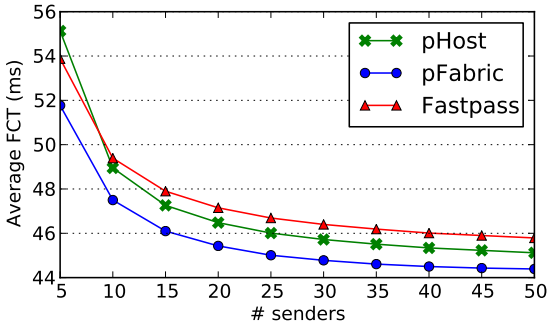
[4]For small buffer sizes ($<$ 36kB) pFabric's performance degrades if we use the default values for its parameters (initial congestion window and retransmission timeout). Hence, for each buffer size, we experimented with a range of different parameter settings for pFabric and select the setting that offers the best slowdown.



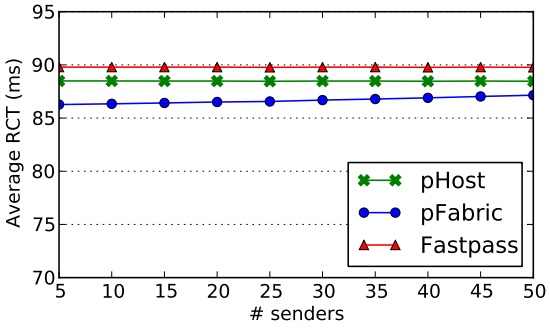
(a) Permutation TM, Figure 5.2 workloads



(b) Permutation TM, synthetic workload



(c) Incast TM, flow completion time



(d) Incast TM, request completion time

Figure 5.9: Performance of the three protocols across various traffic matrices. pHost performs better than pFabric and Fastpass for Permutation TM, and within 5% of pFabric for incast TM.

varying less than 1% over the range of parameters evaluated even with tiny 6.2 KB buffers. We also evaluated variation of throughput with switch buffer sizes, for a workload with 100% long flows and 0.6 network load. The results for that evaluation were very similar with each protocol observing very little impact due to buffer sizes.

5.4.4 Flexibility

Our results so far focused purely on performance goals. However, in addition to performance, datacenter operators must also satisfy policy goals – e.g., ensuring isolation or fairness between different tenants. Compared to pFabric, pHost offers greater flexibility in meeting such policy goals since pHost can implement arbitrary policies for how tokens are granted and consumed at end-hosts. To demonstrate pHost's flexibility, we consider a multi-tenant scenario in which two tenants have different workload characteristics and the operator would like the two tenants to fairly share network bandwidth while allowing each tenant to optimize for slowdown within its share. To achieve this in pHost, we configure pHost's token selection mechanism and packet priority assignment to enforce fairness between the two tenants. This is a minor change: we replace the

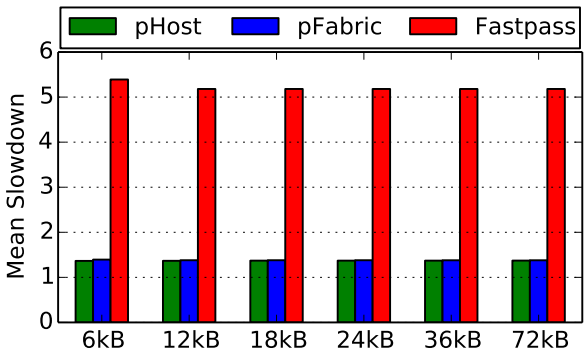


Figure 5.10: Both pHost and pFabric perform well even with tiny buffer sizes. Moreover, the performance of all the three protocols remains consistent across a wide range of switch buffer sizes.

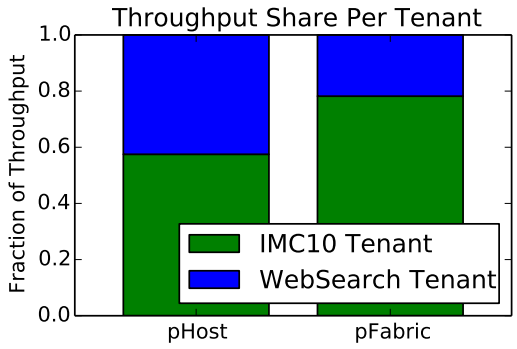


Figure 5.11: pHost, by decoupling flow scheduling from the network fabric, makes is easy to implement diverse policy goals (*e.g.*, fairness in a multi-tenant scenario). In this figure, one tenant gets greater throughput with pFabric (for reasons discussed in §5.4.4), while the throughput is more fairly allocated using pHost.

SRPT priority function with one that does SRPT for flows within a tenant, but enforces that the tenant with fewer bytes scheduled so far should be prioritized. Additionally we turn off data packet priorities (all packets go at the same priority) and remove "free token".

We evaluate a scenario in which one tenant's workload uses the IMC10 trace, while the other tenant's workload uses the Web Search trace. Both the tenants inject the flows in their trace at the beginning of the simulation and we measure the throughput each tenant achieves. Figure 5.11 plots how the overall throughput of the network is shared between the two tenants in pFabric *vs.* pHost.

We see that pFabric allows the IMC10 tenant to achieve significantly higher throughput than the Web Search tenant. This is expected because the IMC10 workload has shorter flows and a smaller mean flow size than the Web Search workload (see Figure 5.2) and hence pFabric implicitly gives the IMC10 tenant higher priority. In contrast, with pHost, the two tenants see similar throughput.

5.5 Related Work

Our work is related to two key network transport designs proposed recently: pFabric [16] and Fastpass [104]. pFabric is a distributed transport mechanism that achieves near-optimal performance in terms of flow completion times; however, pFabric requires specialized hardward that embeds a specific scheduling policy within the network fabric. This approach not only has the disadvantage of requiring specialized network hardware, but also limits generality — the scheduling algorithm cannot be altered to achieve diverse policy goals. Fastpass aims at generality using commodity network fabric along with a centralized scheduler, but loses many of pFabric's performance benefits. pHost achieves the best of the two worlds: the near-optimal performance of pFabric, and the commodity network design of Fastpass.

We compare and contrast pHost design against other rate control and flow scheduling mechanisms below.

Rate Control in Datacenters. Several recent proposals in datacenter transport designs use rate control to achieve various performance goals, such as DCTCP [15], D^2TCP [131], D^3 [137], PDQ [65], PIAS [27], PASE [91]. Specifically, DCTCP uses rate control (via explicit network feedback) to minimize end-to-end latency for short flows. D^2TCP and D^3 use rate control to maximize the number of flows that can meet their respective deadlines. PDQ has goals similar to pFabric; while a radically different approach, PDQ has limitations similar to pFabric — it requires a complicated specialized network fabric that implements PDQ switches. While interesting, all the above designs lose the performance benefits of pFabric [16] either for short flows, or for long flows; moreover, many of these designs require specialized network hardware similar to pFabric. pHost requires no specialized hardware, no complex rate calculations at network switches, no centralized global scheduler and no explicit network feedback, and yet, performs surprisingly close to pFabric across a wide variety of workloads and traffic matrices.

Flow Scheduling in Datacenters. Hedera [14] performs flow scheduling at coarse granularities by assigning different paths to large flows to avoid collision. Hedera improves long flow performance, but ignores short flows that may require careful scheduling to meet performance goals when competing with long flows. Mordia [105] schedules at finer granularity ($\sim 100\mu$s), but may also suffer from performance issues for short flows. Indeed, 100μs corresponds to the time to transmit a $\sim$ 121KB flow in a datacenter with 10Gbps access link capacity. In the Data Mining trace, about 80% flows are smaller than that. Fastpass achieves superior performance by performing per-packet scheduling. However, the main disadvantage of Fastpass is the centralized scheduler that leads to performance degradation for short flows (as shown in §5.4). Specifically, Fastpass schedules an epoch of 8 packets($\sim 10\mu$s) in order to reduce the scheduling and signaling overhead. So no preemption can happen once the 8 packets are scheduled, which fundamentally limits the performance of flows that are smaller than 8 packets. pHost also performs per-packet scheduling but avoids the scalability and performance issues of Fastpass using a completely distributed scheduling at the end hosts.

5.6 Conclusion

There has been tremendous recent work on optimizing flow performance in datacenter networks. The state-of-the-art transport layer design is pFabric, that achieves near-optimal performance but requires specialized network hardware that embeds a specific scheduling policy within the network fabric. We presented pHost, a new datacenter transport design that decouples scheduling policies from the network fabric and performs distributed *per-packet* scheduling of flows using the *end-hosts*. pHost is simple — it requires no specialized network fabric, no complex computations in the network fabric, no centralized scheduler and no explicit network feedback — and yet, achieves performance surprisingly close to pFabric across all the evaluated workloads and network configurations.

Chapter 6

Future Work and Conclusion

Future datacenters will continue to evolve to adapt to new technologies and workloads. As we have discussed, we observe multiple trends in datacenters – serverless computing, resource disaggregation, and priority-based congestion control. We envision that future research work could focus on the following aspects.

Scalable Metadata Coordinator in Savanna. Current Savanna implementation assumes that the Metadata Coordinator is on a single machine, which could be a single point of failure. However, there is no reason that we have to place this functionality on one machine. We have discussed in §3.3, using a RSM (Replicated State Machine) could potentially provide fault tolerance. Besides replicating the metadata coordinator for fault tolerance, the Metadata Coordinator can be sharded using distributed consensus protocols for better scaling property.

Serverless Function Checkpointing. Serverless frameworks often have execution time limitations so that the cloud vendor can easily schedule them. However, this could make it harder for application programmers to write their serverless functions. By using container checkpointing tools such as CRIU [1], we can checkpoint a function when it is running close to the time limit and restore it on other machines. This is akin to process migration in a multicore environment.

POSIX Compliant Savanna. Current Savanna implementation requires programmers to adopt a new API. To be more backward compatible, we can provide Savanna API using the FUSE [2] interface so that Savanna can be mounted as a normal Linux partition.

Efficient Swap. As we have mentioned in §4.5, swap latency introduced by the OS page fault handler could be a significant source of overhead. Current block device driver assumes hard disk as the underlying storage and hence it optimizes the I/O by batching and issuing asynchronous I/O requests. However, when the backing storage is memory, batching and asynchronous I/O could negatively impact the performance. Therefore, a new page fault handling mechanism that can

[1] https://criu.org/

[2] https://github.com/libfuse/libfuse

swap to remote memory more efficiently is necessary for future disaggregated datacenters.

Serverless computing with remote memory. For better resource provisioning, serverless frameworks often require a function to provide a resource limit, such as number of cores and memory limit. Exceeding the memory limit will lead to function failure. If the function could use remote memory, the memory limitation could be relaxed. We believe this would be an useful way to improve the memory utilization in a datacenter.

GPU/FPGA/ASIC Disaggregation. Accelerators such as GPU, FPGA and ASIC are common in modern datacenters. It would be very useful if they can be decoupled from the servers. For example, attaching a GPU at runtime to execute some matrix multiplication workload and detach it after use could effectively increase GPU utilization inside a datacenter. In fact, Amazon is already providing the Amazon EC2 Elastic GPU service that allows an user to attach a remote GPU when launching a EC2 instance.

As a summary, we study how to improve current datacenters by re-architecting the software stack, hardware stack, and network stack. At the software stack, we build Savanna, an extension to serverless framework that provides high performance, function level transaction and automatic fail recovery. At the hardware stack, we evaluate the network requirements for resource disaggregation, and find that using commodity network equipment is sufficient for disaggregation. At the network layer, we design pHost, which provides close-to-optimal flow completion time performance without using specialized hardware.

Bibliography

[1] Amazon EFS Performance. https://goo.gl/3SBgt8.

[2] Announcing Databricks Serverless. https://goo.gl/dahm4h.

[3] Building a Serverless Machine Learning Mode. https://goo.gl/52FA6f.

[4] Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[5] CISCO: Per packet load balancing. http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/pplb.html.

[6] Google Cloud: Cloud Locations.

[7] How to Deploy Deep Learning Models with AWS Lambda and Tensorflow. https://goo.gl/LnKug3.

[8] Internet World Stats. https://www.internetworldstats.com/stats.htm.

[9] Single Root I/O Virtualization (SR-IOV). https://docs.microsoft.com/en-us/windows-hardware/drivers/network/single-root-i-o-virtualization--sr-iov-.

[10] Understanding Retry Behavior. https://goo.gl/DBhf8m.

[11] A. Agache and C. Raiciu. GRIN: Utilizing the empty half of full bisection networks. In *Proc. of HotNets*, 2012.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM 2008.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM*, 2008.

[14] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI*, 2010.

[15] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). SIGCOMM 2010.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. SIGCOMM 2013.

[17] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS)*, 11(4):319–352, 1993.

[18] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in spark. SIGMOD 2015.

[19] K. Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. FAST 2014.

[20] Amazon Athena. `https://aws.amazon.com/athena/`.

[21] AWS Internet of Things. `https://aws.amazon.com/iot/`.

[22] AWS Lambda. `https://aws.amazon.com/lambda/`.

[23] Amazon VPC. `https://aws.amazon.com/vpc/`.

[24] Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[25] Azure Blob Storage. `https://azure.microsoft.com/en-us/services/storage/blobs/`.

[26] Azure Storage Scalability and Performance Targets. `https://goo.gl/UNv6B4`.

[27] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. of NSDI*, 2015.

[28] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7:181–192, 2013.

[29] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, not CAP: Towards highly available transactions. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 24–24. USENIX Association, 2013.

[30] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD Conference*, 2013.

[31] N. Bansal and M. Harchol-Balter. *Analysis of SRPT scheduling: Investigating unfairness*, volume 29. ACM, 2001.

[32] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[33] Berkeley Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.

[34] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. PPOPP 1990.

[35] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of IMC*, 2010.

[36] Big Data System research: Trends and Challenges. `http://goo.gl/38qr1O`.

[37] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated Network Interfaces for High-bandwidth TCP/IP. ASPLOS 2006.

[38] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. In *ACM Computing Surveys*, 2005.

[39] AWS SDK for Python. `https://aws.amazon.com/sdk-for-python/`.

[40] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. OSDI 2006.

[41] J. Cheney, L. Chiticariu, W.-C. Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 2009.

[42] 100G CLR4 White Paper. `http://www.intel.com/content/www/us/en/research/intel-labs-clr4-white-paper.html`.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[44] P. Costa, H. Ballani, K. Razavi, and I. Kash. R2C2: A Network Stack for Rack-scale Computers. SIGCOMM 2015.

[45] Using AWS CodePipeline, AWS CodeBuild, and AWS Lambda for Serverless Automated UI Testing. `https://goo.gl/LphxJP`.

[46] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *Proc. of IEEE INFOCOM*, 2013.

[47] Docker. `https://www.docker.com/`.

[48] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. NSDI 2014.

[49] Bandwidth Growth and The Next Speed of Ethernet. `http://goo.gl/C5lovt`.

[50] Facebook Disaggregated Rack. `http://goo.gl/6h2Ut`.

[51] Apache Flink. `https://flink.apache.org/`.

[52] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, 2017.

[53] Friendster Social Network. `https://snap.stanford.edu/data/com-Friendster.html`.

[54] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. OSDI 2016.

[55] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. CoNEXT 2015.

[56] P. X. Gao, V. Shankar, W. Zhang, Q. Wu, A. Panda, S. Ratnasamy, B. Recht, and S. Shenker. Savanna: An Architectural Extension for Serverless Computing. UC Berkeley.

[57] Google Cloud Storage. `https://cloud.google.com/storage/`.

[58] Google Cloud Functions. `https://cloud.google.com/functions/`.

[59] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[60] A. Greenberg. SDN for the Cloud. SIGCOMM 2015.

[61] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. ACM SIGCOMM CCR 2009.

[62] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. SIGCOMM 2009.

[63] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. HotNets 2013.

[64] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda.

[65] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proc. of SIGCOMM*, 2012.

[66] D. R. Horn, K. Elkabany, C. Lesniewski-Laas, and K. Winstein. The Design, Implementation, and Deployment of a System to Transparently Compress Hundreds of Petabytes of Image Files for a File-Storage Service. NSDI 2017.

[67] High Throughput Computing Data Center Architecture. `http://www.huawei.com/ilink/en/download/HW_349607`.

[68] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. USENIX ATC 2010.

[69] InfiniBand. `http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband`.

[70] Here's How Many Cores Intel Corporation's Future 14-Nanometer Server Processors Will Have. `http://goo.gl/y2nWOR`.

[71] Intel Performance Counter Monitor. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[72] Intel RSA. `http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html`.

[73] Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org/`.

[74] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys 2007.

[75] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed computing for the 99%. SoCC 2017.

[76] Apache Kafka. `https://kafka.apache.org/`.

[77] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. SIGCOMM 2014.

[78] S. Kumar. Petabit Switch Fabric Design. Master's thesis, EECS Department, University of California, Berkeley, 2015.

[79] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *SOSP*, 2011.

[80] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. SoCC 2014.

[81] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. TOCS 1989.

[82] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. OSDI 2014.

[83] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-Value Storage. NSDI 2014.

[84] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. ISCA 2009.

[85] K. Lim, Y. Turner, J. R. Santos, A. Auyoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. HPCA 2012.

[86] Linux Containers. `https://linuxcontainers.org/`.

[87] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? HotOS 2015.

[88] Memcached - A Distributed Memory Object Caching System. `http://memcached.org`.

[89] Memristor. `http://www.memristor.org/reference/research/13/what-are-memristors`.

[90] M. Mitzenmacher. The power of two choices in randomized load balancing. TPDS 2001.

[91] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proc. of SIGCOMM*, 2014.

[92] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. SOSP 2013.

[93] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant Software Distributed Shared Memory. USENIX ATC 2015.

[94] Netflix Rating Trace. `http://www.select.cs.cmu.edu/code/graphlab/datasets/`.

[95] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. ASPLOS 2014.

[96] Huawei NUWA. `http://nuwabox.com`.

[97] Graphics Processing Unit. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

[98] Non-Volatile Random Access Memory. https://en.wikipedia.org/wiki/Non-volatile_random-access_memory.

[99] Intel Omnipath. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[100] Apache OpenWhisk. https://openwhisk.incubator.apache.org/.

[101] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. TOCS 2015.

[102] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. NSDI 2015.

[103] O. O'Malley. Terabyte sort on apache hadoop. pages 1–3, 2008.

[104] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. SIGCOMM 2014.

[105] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proc. of SIGCOMM*, 2013.

[106] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The tcp outcast problem: exposing unfairness in data center networks. In *NSDI 2012*.

[107] C. Raiciu, M. Ionescu, and D. Niculescu. Opening up black box networks with CloudTalk. In *4th USENIX Conference on Hot Topics in Cloud Ccomputing*, 2012.

[108] P. S. Rao and G. Porter. Is Memory Disaggregation Feasible?: A Case Study with Spark SQL. ANCS 2016.

[109] J. Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[110] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. USENIX ATC 2012.

[111] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. Its Time for Low Latency. HotOS 2011.

[112] Amazon S3. https://aws.amazon.com/s3/.

[113] Request Rate and Performance Considerations. https://goo.gl/SdypS3.

[114] S3 Support in Apache Hadoop. https://wiki.apache.org/hadoop/AmazonS3.

[115] SeaMicro Technology Overview. http://seamicro.com/sites/default/files/SM_TO01_64_v2.5.pdf.

[116] serverless.com. https://serverless.com/.

[117] Serverless IoT Backends. https://goo.gl/DtHt2q.

[118] Serverless Web Applications. https://goo.gl/kjNXo6.

[119] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Nfs version 4 protocol, December 2000. RFC3010.

[120] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. SIGCOMM 2015.

[121] R. P. Singh, P. X. Gao, and D. J. Lizotte. On hourly home peak load prediction. SmartGridComm 2012.

[122] Sort Benchmark. http://sortbenchmark.org.

[123] K. Sudan, S. Balakrishnan, S. Lie, M. Xu, D. Mallick, G. Lauterbach, and R. Balasubramonian. A Novel System Architecture for Web Scale Applications Using Lightweight CPUs and Virtualized I/O. HPCA 2013.

[124] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, et al. Single-chip Microprocessor that Communicates Directly Using Light. *Nature 2015*.

[125] "tcpdump". http://www.tcpdump.org.

[126] HP The Machine. http://www.hpl.hp.com/research/systems-research/themachine/.

[127] A look at The Machine. https://lwn.net/Articles/655437/.

[128] Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow.

[129] S. Tu, R. Roelofs, S. Venkataraman, and B. Recht. Large scale kernel learning using block coordinate descent. *arXiv preprint arXiv:1602.05310*, 2016.

[130] Twitter Stream Data. https://goo.gl/Svu8BV.

[131] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.

[132] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. RAID 2009.

[133] T. Wagner. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/ ,Retrieved09/14/2017, 2014.

[134] M. Walraed-Sullivan, J. Padhye, and D. A. Maltz. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. HotNets-XIII.

[135] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[136] Wikipedia Dump. https://dumps.wikimedia.org/.

[137] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proc. of SIGCOMM*, 2011.

[138] H. Wu, Z. Feng, C. Guo, and Y. Zhang. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM transactions on networking*.

[139] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News, March 1995*.

[140] Intel Ethernet Converged Network Adapter XL710 10/40 GbE. http://www.intel.com/content/www/us/en/network-adapters/converged-network-adapters/ethernet-xl710-brief.html.

[141] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate ddos flooding attacks. In *IEEE Symposium on Security and Privacy*, 2004.

[142] M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI 2012.

[143] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. SOSP 2013.

[144] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP*, 2015.

[145] Bandwidth: a memory bandwidth benchmark. http://zsmith.co/bandwidth.html.