UNIVERSITY OF CALIFORNIA, SAN DIEGO

Algorithms for Measuring and Enhancing Distributed Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Frank C. Uyeda

Committee in charge:

Professor George Varghese, Chair
Professor Gert Lanckriet
Professor Lev Manovich
Professor Yannis Papakonstantinou
Professor Amin Vahdat

2011

The dissertation of Frank C. Uyeda is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

TABLE OF CONTENTS

Signature Page . . . iii

Table of Contents . . . iv

List of Figures . . . vi

List of Tables . . . viii

Acknowledgements . . . ix

Vita . . . xii

Abstract of the Dissertation . . . xiii

Chapter 1 Introduction . . . 1
1.1 Contributions . . . 3

Chapter 2 Set Synchronization with Prior Context using Difference Digests . . . 6
2.1 Model and Related Work . . . 9
2.2 Algorithms . . . 11
2.2.1 Invertible Bloom Filter . . . 11
2.2.2 Strata Estimator . . . 16
2.3 Analysis . . . 20
2.4 The KeyDiff System . . . 23
2.5 Evaluation . . . 25
2.5.1 Tuning the IBF . . . 26
2.5.2 Tuning the Strata Estimator . . . 28
2.5.3 Difference Digest vs. ARTree . . . 34
2.5.4 KeyDiff Performance . . . 35
2.6 Conclusions . . . 41

Chapter 3 Efficiently Measuring Bandwidth at All Time Scales . . . 42
3.1 Related Work . . . 47
3.2 Algorithms . . . 48
3.2.1 Dynamic Bucket Merge . . . 49
3.2.2 Exponential Bucketing . . . 56
3.2.3 Culprit Identification . . . 59
3.3 Evaluation . . . 60
3.3.1 Measurement Accuracy . . . 60
3.3.2 Performance Overhead . . . 68
3.3.3 Evaluation Summary . . . 69
3.4 System Implications . . . 70

3.5 Conclusions 71

Chapter 4 Evaluating the Efficacy of Software-Based Traffic Pacing 76
4.1 Software Token Buckets and their Problems 78
4.2 Tools for Measuring the Effectiveness of Pacing in Software 82
4.2.1 BandwidthTracker 84
4.2.2 BufferSim 86
4.2.3 Linux Kernel Implementation of Tools 88
4.3 Evaluating Linux Token Bucket Filter 89
4.3.1 Bandwidth Problems with TBF 90
4.3.2 Average Bandwidth after Fixing `tc` 92
4.3.3 Balancing Bandwidth and Buffering 94
4.3.4 Dissecting Timer Settings in TBF 97
4.4 Improving the TBF code 101
4.4.1 Avoid Timer Readjustment 101
4.4.2 Send Early 102
4.5 Related Work 104
4.6 Conclusion 108

Chapter 5 Conclusions 110

Bibliography 113

LIST OF FIGURES

Figure 2.1: IBF Encode . . . 14
Figure 2.2: IBF Subtract . . . 15
Figure 2.3: IBF Decode . . . 18
Figure 2.4: Partitioning Elements for the Strata Estimator . . . 19
Figure 2.5: Estimating the Size of the Set Difference with a Strata Estimator . . 21
Figure 2.6: KeyDiff Service . . . 24
Figure 2.7: Rate of successful IBF decode . . . 26
Figure 2.8: Probability of successful decoding for IBF with varying hash counts 27
Figure 2.9: IBF cells required for decoding with 99% certainty . . . 28
Figure 2.10: Overhead required for IBF decoding with 99% certainty . . . 29
Figure 2.11: Strata Estimator performance versus strata size . . . 30
Figure 2.12: Comparison of Estimators . . . 33
Figure 2.13: Data transmission required by ART and Difference Digests to recover 95% and 100% of the set difference, respectively, with 99% reliability. . . . 34
Figure 2.14: Time to run KeyDiff . . . 40

Figure 3.1: Example Monitoring Deployment . . . 47
Figure 3.2: Dynamic Bucket Merge with 4 buckets . . . 52
Figure 3.3: Exponential Bucketing Example . . . 58
Figure 3.4: Relative error for DBM−mr algorithm shown for the 400 μs time scale with a varying number of buckets . . . 64
Figure 3.5: Visualization of events from a 2 second aggregation period overlaid with the output of DBM−mr using 9 buckets and a 4 msec measurement time scale. . . . 66
Figure 3.6: DBM Visualization Example . . . 67
Figure 3.7: Average relative error for DBM and EXPB on the TritonSort trace . . 68

Figure 4.1: Pseudocode for the Linux Token Bucket algorithm . . . 80
Figure 4.2: Bandwidth loss caused by inaccurate timers. . . . 81
Figure 4.3: A probabilistic model for bandwidth degradation in the presence of inaccurate timers in software pacing. . . . 82
Figure 4.4: Illustrating Recalibration . . . 83
Figure 4.5: Recalibration Pseudocode for the BandwidthTracker algorithm. . . 85
Figure 4.6: Pseudocode for the Ideal Buffer Simulation. . . . 87
Figure 4.7: Recalibration Pseudocode for the BufferSim algorithm. . . . 88
Figure 4.8: Software organization . . . 89
Figure 4.9: Observed bandwidth deviates from the specified rate . . . 90
Figure 4.10: TBF results at 5 Gbps with different packet sizes . . . 91
Figure 4.11: Specified versus observed bandwidth after fixing precision bug . . . 93

Figure 4.12: Average bandwidth versus packet sizes after correcting the precision bug for a fixed target rate of 5Gbps. 94
Figure 4.13: The average bandwidth and maximum buffer occupancy with a target rate of 5Gbps for various bucket sizes. 95
Figure 4.14: Variability of observed bandwidth for various bucket sizes in the face of interference from high volume of UDP receive traffic. . . . 96
Figure 4.15: Performance of chained token buckets 97
Figure 4.16: Percentage of time lost due to overflowing token bucket versus the size of the token bucket 98
Figure 4.17: Distribution of timer expirations with and without interfering UDP receive load 99
Figure 4.18: Distribution of timer error 100
Figure 4.19: Distribution of timer error for long and short timer expirations . . . 101
Figure 4.20: Updated pseudocode for `serviceQueue` from Figure 4.1 to avoid timer readjustment. 103
Figure 4.21: Updated pseudocode for `serviceQueue` from Figure 4.1 for sending early to avoid setting short timers. 105
Figure 4.22: Bandwidth vs. bucket size for our two improvements - sending early (1024nsec delta) and eliminating timer resets. 107

LIST OF TABLES

Table 2.1: Latency of KeyDiff operations 37

Table 3.1: Evaluation of memory vs. accuracy for TritonSort trace 73
Table 3.2: Evaluation of memory vs. accuracy for rsync trace 74
Table 3.3: Network performance overhead 74
Table 3.4: Time to transfer 1GB file using `scp` 75

Table 4.1: Measurement of TBF improvements 106

ACKNOWLEDGEMENTS

I would first like to offer my heart-felt thanks to my advisor, George Varghese, for his direction and insights in guiding my development as a researcher. Thank you for your support as I considered entering the Ph.D. program at UCSD, for your energy and enthusiasm in our discussions, for your patience when I struggled to make progress, for allowing me to change directions with my research, and for praying for me. I certainly could not have done this without your support.

I would also like to thank my Masters advisors, Amin Vahdat and Andrew Chien. To Andrew, thank you for first giving me the opportunity to do research as an undergrad. To Amin, thank you for always making time to talk, for being genuinely interested in how I was doing, for giving your honest opinion, and for encouraging me along the way.

Thank you to my all my collaborators, without whose help and insights I certainly would not have made it so far: Mike Goodrich & David Eppstein from UC Irvine, Subhash Suri & Luca Foschini from UC Santa Barbara, Ant Rowstron from MSR Cambridge, Dejan Kostic & Simon Schubert from EPFL, John Douceur & Jay Lorch from MSR Redmond, Jeff Pang from CMU, and Diwaker Gupta, Ryan Braud, Justin Burke, Nut Taesombut & Huaxia Xia from UCSD.

Thank you to my officemates in CSE 3142: Dejan Kostic, Justin Burke, Ming Kawaguchi, Gjergji Zyba, Terry Lam, Ryan Roemer, Kevin Bauer, Siva Radhakrishnan, Erik Buchanan, Bhanu Vattikonda, Malveeka Tewari, and Michael Lee. Thank you for sanity checking my crazy ideas, proof reading my papers, introducing me to backgammon, taking breaks each day to play foosball, and just chatting about life.

Thank you also to my officemates from the Concurrent Systems and Architecture Group: Huaxia Xia, Nut Taesombut, Yang-suk Kee, and Justin Burke. Thank you for making me feel welcome when I first joined you as a clueless undergrad and for helping me learn the ropes.

Thank you to the other graduate students and post-docs who have shared in my time at UCSD: Barry Demchak, Yuvraj Agarwal, Michael Vrable, Diwaker Gupta, Alvin AuYoung, Patrick Verkaik, Marti Motoyama, Chris Kanich, Meg Walraed-Sullivan, Nathan Farrington, John McCullough, Alex Rasmussen, Kevin Webb, George Porter, Kirill Levchenko, and Ken Yocum.

In addition, the Systems and Networking faculty deserve recognition for fostering an environment of where we could all collaborate and learn together. Thanks to Alex Snoeren, Geoff Voelker, Amin Vahdat, Stefan Savage, George Varghese, Joe Pasquale & Keith Marzullo for making that possible.

I would also like to thank my mentors from my two summers at Microsoft Research, John Douceur and Ant Rowstron. Thank you for your guidance and the fantastic experience of working at one of the world's premier research labs.

Thank you to the support staff and systems administrators in the Systems and Networking Group and the Center for Networked Systems: Marvin McNett, Chris Edwards, Brian Kantor, Cindy Moore, and Kathy Krane. Also, thank you to the department staff who have been very helpful throughout the years: Julie Conner, Viera Kair, Dave Wargo, Bill Young, Kim Graves, Cheryl Hile and Michelle Panik.

I would also like to express my gratitude to all my teammates and coaches, both past and present, from the UCSD Triathlon Team. It has truly been a joy and pleasure to train and race with all of you. I'm very thankful for all the conversations during long runs and rides, for all the things that you've taught me and inspired me to do, and for the friendship and camaraderie we've shared.

Thank you to my family for all of your love, support, and prayers over the years. To Mom and Dad, thank you for all the council you've given as I've figured out my course in life. To my wife, Kellie, thank you for loving me and encouraging me through my ups and downs. Thank you for affirming me when I get discouraged, for jump-starting my progress when procrastination gets the better of me, and for joining me as we share all the blessings and goodness of life together.

Finally, I must recognize and thank God for His abundant provision and blessing over the years. His goodness and love continue to amaze me and His presence in my life has been my source of peace and strength.

Chapter 2, in part, is a reprint of the material as it will appear in the article "Set Synchronization without Prior Context using Difference Digests" in the Proceedings of the Special Interest Group on Data Communications (SIGCOMM), Toronto, ON, Canada, August 2011. David Eppstein, Michael T. Goodrich, Frank Uyeda, George Varghese.

Chapter 3, in part, is a reprint of the material as it appears in the article "Efficiently Measuring Bandwidth at All Time Scales" in the Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, March 2011. Frank Uyeda, Luca Foschini, Subhash Suri, George Varghese.

Chapter 4, in part, is a reprint of the material being prepared for submission for publication. "How Effective is Software Pacing?". Frank Uyeda, Amin Vahdat, George Varghese.

VITA

2004	Bachelor of Science, Computer Engineering, *summa cum laude*, University of California, San Diego
2006	Master of Science, Computer Science, University of California, San Diego
2009-2010	Teaching Assistant, University of California, San Diego
2009	Instructor, University of California, San Diego
2011	Doctor of Philosophy, Computer Science, University of California, San Diego

PUBLICATIONS

David Eppstein, Michael T. Goodrich, Frank Uyeda, George Varghese. "Set Synchronization without Prior Context using Difference Digests", *To appear at the Special Interest Group on Data Communications (SIGCOMM)*, Toronto, ON, Canada, August 2011.

Frank Uyeda, Luca Foschini, Subhash Suri, George Varghese. "Efficiently Measuring Bandwidth at All Time Scales", *In Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2011.

Simon Schubert, Frank Uyeda, Nedeljko Vasic, Dejan Kostic. "Bandwidth Adaptation in Streaming Overlays", *In Proceedings of the 2nd International Conference on COMmunication Systems and NETworkS (COMSNETS)*, Bangalora, India, 2010.

Frank Uyeda, Diwaker Gupta, Amin Vahdat, George Varghese, "GrassRoots: Socially-Driven Web Sites for the Masses", *Workshop on Online Social Networks (WOSN)*, Barcelona, Spain, August 2009.

John Douceur, Jacob Lorch, Frank Uyeda, Randall Wood, "Enhancing Game-Server AI with Distributed Client Computation", *In Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 31-36, Urbana, IL, June 2007

Jeffrey Pang, Frank Uyeda, Jacob Lorch, "Scaling Peer-to-Peer Games in Low Bandwidth Environments", *In Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS)*, Bellevue, WA, February 2007

Nut Taesombut, Frank Uyeda, Andrew Chien, "The OptIPuter: High Performance, QoS-Guaranteed Network Service for Emerging E-Science Applications", *IEEE Communications Magazine*, Vol 44(5), Pages 38-45, May 2006.

ABSTRACT OF THE DISSERTATION

Algorithms for Measuring and Enhancing Distributed Systems

by

Frank C. Uyeda

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor George Varghese, Chair

The industry-wide movement toward large data centers and cloud computing has brought many economic advantages, but also numerous technical challenges. In this work we describe three software-based contributions towards improving the communication performance of these large-scale distributed systems.

While advances to commodity Ethernet in the data center have increased throughput, efficient application design can produce even more significant speedups. We first describe an efficient, new data structure, called "Difference Digests", that can serve as a building block in distributed-application protocols. Difference Digests identify the elements that differ between two sets using communication and computation overheads proportional to population of the difference. This functionality has many promising ap-

plications, including recovery from network partitions, data synchronization, and data de-duplication.

Beyond more efficient protocols, tuning applications can further improve performance by reducing network congestion. As link speeds increase, measuring bandwidth at fine time scales produces valuable debugging and tuning information, but is complicated by the immense number of packets and the short per-packet processing time. Further, it is unclear in advance which time scales will prove insightful, but storing all measurements incurs significant storage overhead. Thus, we propose two algorithms to summarize streams of fine-grain bandwidth samples and identify bursty traffic behavior. Our implementation records bandwidth information at speeds up to 10 Gbps while decreasing storage costs by orders of magnitude. After the measurement, bandwidth statistics can be generated for arbitrary time scales down to microseconds, allowing users to identify short-lived phenomena affecting their application's performance.

While network capacity has increased, maintaining low latency for time-sensitive flows remains challenging. One approach is to minimize in-network buffering by controlling flows with software-based pacing at the end host. However, these mechanisms are unproven at the Gigabit speeds found in data centers. Hence, we demonstrate measurement tools to evaluate the bandwidth and "burstiness" of one such mechanism, the Linux Token Bucket Filter (TBF). We find that TBF struggles to scale to multi-Gigabit speeds due to Linux's inability to reliably service timers with micro-second accuracy. To address this limitation, we describe two enhancements, which improve software-based pacing at speeds up to 10 Gbps while minimizing bursts.

Chapter 1

Introduction

Computing has undergone a fundamental shift in the past decade. While the 1990's were marked by the rise of personal computers and desktop applications, recent years have seen a shift toward ubiquitous access to large websites and services provided "in the cloud". Companies such as Google, Yahoo!, and Facebook boast hundreds of millions of users [2] who access a plethora of services through their web browsers, including web search, messaging, social networking, maps, and photo sharing. Offered at minimal to no cost, cloud services have replaced many desktop applications by enabling new functionality and improving existing features.

In order to support these services, there has been an industry-wide trend toward consolidating computing resources into data centers. The thousands of machines housed and operated within a data center are connected via a high-speed networking infrastructure and run numerous distributed applications, such as web servers, web caches, database servers, and large-scale data processing jobs. These applications serve as the foundation, which enables companies to run their large-scale websites and cloud computing services.

In order to exploit economies of scale, many companies now construct massive data centers. Such data centers might now contain up to one hundred of thousand of machines. Building facilities at this scale allows operators to cut operating costs in maintainance, power conversion, cooling, and personel . However, building such large distributed systems also creates numerous technical challenges.

One of the key bottlenecks in achieving high performance in large data centers

is the network. While modern deployments utilize high-speed links running at 10 Gbps, network contention still remains a problem. Several factors contribute to the network bottleneck. First, data center networks are built in a hierarchical topology with over-subscribed links in the center of the network. This means that connections between devices in the network's core do not have sufficient capacity to allow full bi-section bandwidth between the racks of machines that they connect. In such a scenario, the possible bandwidth demand may exceed the capacity of the link by a factor of 10 or more. Even as 40 and 100 Gbps links begin to appear, this problem is not easily remedied due to Ethernet's spanning tree design and to the growing number of machines in the network.

Second, many applications share the network and all compete for the limited bandwidth resources. While applications are written to achieve high-performance, developers can rarely anticipate environment where their applications will be deployed or which other applications they will be co-located with. Thus, it is up to administrators to tune application performance to achieve a good trade off between the demands of all applications in the data center.

Varying application demands constitue the third challenge. While some applications are primarily constrained by throughput, others require consistent, low-latency connections. These objectives directly conflict as achieving high throughput typically involves buffering data in the network so that links can achieve high utilization. However, in-network buffering incurs extra delay for flows, include those which are latency sensitive.

Finally, the use of commodity networking equipment hinders application performance. For data-center operators who are constructing massive facilities, there are significant savings associated with provisioning commodity networking hardware. However, to control costs, these devices are designed with a minimal quantities of expensive high-speed memory for packet buffering. These shallow buffers make commodity switches more prone to drop packets, which causes costly retransmissions.

Two main avenues may be pursued to improve the communication performance of large-scale distributed systems. First, designing more efficient application protocols alieviates presure on networking resources. Thus, the severity of network congestion

can be mitigated by reducing the total load. While sophisticated, new protocols may be employed, theoretical bounds and the scope of current understanding limit the achievable improvements.

Second, coordinating communication patterns can reduce network congestion. Toward this end, administrators can tune applications to decrease contention, deploy new protocols that react to congestion, or police traffic entering the network to avoid congestion. However, the need for hardware support to achieve good performance at multi-Gigabit speeds presents a major challenge to these approaches. Specifically, the cost of designing, fabricating and deploying custom hardware blunts the appeal of many of these approaches.

In this dissertation we challenge the idea that customized hardware is required to improve the performance of distributed applications on high-speed networks. We argue that software-based solutions can improve performance and demonstrate this through the design, implementation, and evaluation of three software systems.

1.1 Contributions

In this work, we seek to enhance applications' communication performance by increasing communication efficiency for certain common operations, providing effective bandwidth monitoring to tune and debug communication performance, and improving traffic regulation at the end hosts to decrease latency caused by in-network buffering.

Even as the communication performance in data centers increases due to faster hardware and better traffic engineering, efficient protocol design can result in even more significant application speedups. In Chapter 2, we will describe a new data structure, called "Difference Digests", which efficiently solves the "set difference" problem and can be used to improve distributed protocols for data synchronization, data de-duplication, and recovery from network partitions.

While other solutions for set difference exist [18, 37], Difference Digests are more efficient — identifying the elements that differ between two data sets with communication and computation overheads proportional to the population of the difference. We achieve this by extending the Invertible Bloom Filter (IBF) [23, 28] structure with

a subtraction operator and applying it to the set difference problem. However, a key hurdle in using IBF's is appropriately sizing them to achieve good communication efficiency. We address this problem through a second crucial component of the Difference Digest called a Strata Estimator, which estimates the *size of the difference* between two sets. In our evaluation, we find that this estimator is much more accurate for small set differences than approaches like Min-Wise Sketches [15, 16]. Finally, we describe the implementation and evaluation of a generic KeyDiff service, which uses Difference Digest to allow applications to synchronize objects of any kind.

Next, we address the problem of detecting bursty traffic patterns in deployed systems so that administrators can take corrective measures. Increases in network capacity, which dramatically increase the volume of packets and decrease per-packet processing time, complicate this task. In the data-center environment, existing measurement techniques either incur high storage costs [10, 1], or are not well suited for short-lived burst detection [9, 35]. While summarization techniques exist for a small number of preset sampling resolutions [40], it is often not clear in advance which time scales will exhibit anomalous behavior or that those time scales will remain relevant over time. The algorithmic challenge is to support *a posteriori* queries without retaining the entire trace or keeping state for all time scales.

To address these problems, we propose a generalized solution that can generate bandwidth statistics and can detect bursts at *all time scales*. In Chapter 3, we introduce two aggregation algorithms, Dynamic Bucket Merge (`DBM`) and Exponential Bucketing (`EXPB`), that compactly summarize streams of fine-grain bandwidth measurements to identify bursty traffic behavior. Our solution records bandwidth information at line rate on links up to 10 Gbps while decreasing storage costs by orders of magnitude. Users can then query for bandwidth statistics at arbitrary time scales down to microseconds, as well as visualize bandwidth behavior over time. Further, we suggest a technique for attributing bursts to individual flows, allow administrators to identify the applications responsible for the bursty behavior.

Finally, in Chapter 4, we explore the performance of software-based traffic pacing at end hosts, a technique that seeks to maintain high throughput while improving worst-case latency by smoothing bandwidth variability and avoiding network conges-

tion. We construct two measurement tools to evaluate the bandwidth and "burstiness" of the Linux Token Bucket Filter (TBF), a commonly used traffic pacer. Building our tools with a simple recalibration paradigm, we are able to accurately measure TBF when run at the multi-Gigabit speeds and under high system load. Based on our experiments, we find that TBF struggles to achieve both high bandwidth and low burstiness at high speeds due to Linux's inability to reliably service timers with microsecond accuracy. To address this limitation, we describe two enhancements, which improve software-based pacing at speeds up to 10 Gbps while minimizing bursts.

Chapter 2

Set Synchronization with Prior Context using Difference Digests

Suppose you met a friend after many years and wanted to share the vast, but similar, music libraries on each of your computers. How could you efficiently determine the size of the difference and communicate the few unique songs?

Such reconciliation between two hosts on a network can be abstracted as a set synchronization problem, where the goal is to efficiently synchronize the sets of keys between two hosts so that each host obtains the union of these sets. This problem is often encountered by storage and compute nodes within and across data centers where efficient solutions are measured by the latency required for synchronization, as well as by the bandwidth, memory, and computation consumed.

A simple solution to the set synchronization problem is to exchange and compare complete lists of each host's data objects; however, this would transfer an amount of information proportional to the total number of objects rather just the unique elements in the two sets. An often-used, alternative approach is to maintain a time-stamped log of updates, together with a record of the time that the users last communicated. When they communicate again, host A can send host B all of the updates since their previous communication, and vice versa. However, this requires stable storage for the logs, and it requires the logging system to be integrated with any system that can change either user's data. Additionally, unless A and B communicate only with each other, they may both have received the same updates from a third user since they last communicated,

leading to redundant communication.

By contrast, in this chapter, we are interested in solutions to the set synchronization problem that do not have these weaknesses. We wish to handle the case when A and B have no (or stale) prior context and can independently receive updates from other nodes. We also wish our algorithm to work on networking devices such as routers without synchronized time stamps or stable storage. To solve these problems, we devise a data structure, called a Difference Digest, that allows two nodes to compute the approximate size of, and the elements belonging to, the difference between two sets in a single round with communication overhead proportional to the size of the difference times the logarithm of the keyspace. Once the elements in the set difference have been identified they can be transmitted between the hosts to synchronize the sets.

We implement and evaluate Difference Digests and show that, for large sets and small set differences, Difference Digests are more efficient than prior approaches, such as Approximate Reconciliation Trees [18]. For example, in sets containing 1 million keys but differing in only 100 elements, Difference Digests require 20KB to reconcile the difference—almost two orders of magnitude less than Approximate Reconciliation Trees. Further, we use Difference Digests to implement a generic KeyDiff service in Linux that runs over TCP and computes the sets of keys that differ between machines. Settings in which this functionality may be applied include:

- *Peer-to-peer:* Peer A and B may receive blocks of a file from other peers and may wish to receive only missing blocks from each other.

- *Partition healing:* When a link-state network partitions, routers in each partition may each obtain some new link-state packets. When the partition heals by a link joining router A and B, both A and B only want to exchange new or changed link-state packets.

- *Synchronizing output:* A search engine may use two independent crawlers with different techniques to harvest URLs but they may have very few URLs that are different. In general, this situation arises when multiple actors in a distributed system are performing the same function for redundancy or coverage but may obtain slightly different results. A second example is when multiple log servers

are used but each may lose different parts of the log.

- *Opportunistic ad hoc networks:* These are often characterized by low bandwidth and intermittent connectivity to other peers. Examples include rescue situations and military vehicles that wish to synchronize data when they are in range.

The main contributions of the chapter are as follows:

- *IBF Subtraction:* The first component of the Difference Digest is an Invertible Bloom Filter or IBF [23, 28]. IBF's were previously used [23] for straggler detection at a single node, to identify items that were inserted and not removed in a stream. We adapt invertible Bloom filters for set reconciliation by defining a new subtraction operator on whole IBF's (as opposed to individual item removal).

- *Strata Estimator:* To compute set differences efficiently, Invertible Bloom filters must be appropriately sized. Thus, a second crucial component of the difference digest is a new Strata Estimator method for estimating the *size of the difference*. We show that this estimator is much more accurate for small set differences (as is common in the final stages of a file dissemination in a P2P network) than Min-Wise Sketches [15, 16] or random projections [31]. Besides being an integral part of the Difference Digest, our Strata Estimator can be used independently to find, for example, which of many peers is most likely to have a missing block.

- *KeyDiff Prototype:* We describe an initial Linux prototype of a generic KeyDiff service based on Difference Digests that applications can use to synchronize objects of any kind.

- *Performance Characterization:* Not surprisingly, the overall system performance of Difference Digests is sensitive to many parameters, such as the size of the difference, the bandwidth available compared to the computation, and the ability to do pre-computation. We characterize the parameter regimes in which Difference Digests outperform other algorithms based on earlier approaches such as Min-Wise hashes and Approximate Reconciliation Trees [18], and show that the best system approach is a combination of Difference Digests with earlier algorithms.

Going forward, we discuss related work in Section 2.1 and present our algorithms for the Invertible Bloom Filter and Strata Estimator components of a Difference Digest in Section 2.2. We evaluate each of these structures in Section 2.5 and conclude in Section 2.6.

2.1 Model and Related Work

We start with a simple model of set synchronization. For two sets S_A, S_B each containing elements from a universe, $\mathbb{U} = [0, u)$, we want to compute the set difference, D_{A-B} and D_{B-A}, where $D_{A-B} = S_A - S_B$ such that for all $s \in D_{A-B}$, $s \in S_A$ and $s \notin S_B$. Likewise, $D_{B-A} = S_B - S_A$. We say that $D = D_{A-B} \cup D_{B-A}$ and $d = |D|$. Note that since $D_{A-B} \cap D_{B-A} = \emptyset$, $d = |D_{A-B}| + |D_{B-A}|$. We assume that S_A and S_B are stored at two distinct hosts and attempt to compute the set difference with minimal computation, storage, and communication.

In the prior work on identifying elements in a set difference, several algorithms have been proposed. The simplest approach requires hosts to exchange a list of identifiers for all elements in their sets and then iteratively scan the local and remote lists to remove common elements. This method requires O($|S_A| + |S_B|$) communication and O($|S_A| \times |S_B|$) time. This run time can be improved to O($|S_A| + |S_B|$) by inserting one list into a hash table, then querying the table with the elements of the second list. The communication overhead can be reduced by exchanging Bloom filters [14] containing the elements of each list. Once in possession of the remote Bloom Filter, each host can query the filter to identify which elements of its set are common between the hosts. One drawback to this method is the risk of false positives when querying if the table has insufficient size. The time cost for this procedure is O($|S_A| + |S_B|$). Incidentally, the Bloom filter approach has been extended with Approximate Reconciliation Trees [18], which requires O($|S_A| + d \log(|S_B|)$) time, and O($|S_B|$) space. However, given S_A and an Approximate Reconciliation Tree for S_B, a host can only compute D_{A-B}, i.e., the elements unique to S_A.

An exact approach to the set-difference problem was proposed by [37]. In this approach, each set is encoded using a linear transformation similar to Reed-Solomon

coding. This approach has the advantage of O(d) communication overhead, but requires O(d^3) time. The time complexity can be improved to O(d) expected time, but requires an interactive decoding process that uses several rounds of communication. Therefore this method is less desirable when the size of the difference can be large or in environments with long communication latencies.

Estimating Set-Difference Size A critical sub-problem related to many of these algorithms (including ours) is an initial estimation of the size of the set difference. The size of the set difference can be estimated by comparing a random sample [31] from each set. This can be done with constant overhead, but accuracy quickly deteriorates when the d is small relative to the size of the sets.

Min-wise sketches [15, 16] can be used to estimate the set-similarity ($r = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$). Min-wise sketches work by selecting k random hash functions $\pi_1, \ldots, \pi_k$ that permute elements within $\mathbb{U}$. Let min($\pi_i(S)$) be the smallest value produced by π_i when run on the elements of S. Then, a Min-wise sketch consists of the k values min($\pi_1(S)$), …, min($\pi_k(S)$). If min($\pi_i(S_A)$) = min($\pi_i(S_B)$) then there exists an element s in both S_A and S_B such that $\pi_i(s) =$ min($\pi_i(S_A)$) = min($\pi_i(S_B)$). Hence, we know that $s \in S_A \cup S_B$ and $s \in S_A \cap S_B$.

Given that all π_i are random permutations, an element $s \in S_A \cup S_B$ will produce $\pi_i(s) = min(\pi_i(S_A \cup S_B))$ with probability $\frac{1}{|S_A \cup S_B|}$. Therefore, if S_A and S_B have a set-similarity r, then we expect that the number of matching cells in M_A and M_B will be $m = rk$. Inversely, given that m cells of M_A and M_B do match, we can estimate that $r = \frac{m}{k}$. Given the set-similarity, we can estimate the difference as $d = \frac{1-r}{1+r}(|S_A| + |S_B|)$. As with random sampling, the accuracy of the Min-wise estimator diminishes for smaller values of k and for relatively small set differences. Similarly, Cormode *et al.* [21] provide a method for dynamic sampling from a data stream to estimate set sizes using a hierarchy of samples, which include summations, counts, and collision detection components. (See also Muthukrishnan [40] for a discussion of these and related approaches.)

While efficiency of set-difference *estimation* for small differences may seem like a minor theoretical detail, it can be important in many contexts. Consider, for instance

the endgame of a P2P file transfer. Imagine that a BitTorrent node has 100 peers, and is missing only the last block of a file. Min-wise or random samples from the 100 peers will not identify the right peer if all peers also have nearly finished downloading (small set difference). On the other hand, sending a Bloom Filter or Approximate Reconciliation Tree takes bandwidth proportional to the number of blocks in file, which can be large. In Section 2.2.2, we will describe a new Strata Estimator that outperforms earlier approaches for small set differences.

2.2 Algorithms

In this section, we describe the two components of the Difference Digest: an Invertible Bloom Filter (IBF) and Strata Estimator. Our first innovation is taking the existing IBF [23, 28] and introducing a subtraction operator in Section 2.2.1 to compute D_{A-B} and D_{B-A} using a single round of communication of size O(d). Encoding a set S into an IBF requires O($|S|$) time, but decoding to recover D_{A-B} and D_{B-A} requires only O(d) time. Our second key innovation is a way of composing several sampled IBF's of fixed size into a new Strata Estimator, which can effectively estimate the size of the set difference using O(log($|\mathbb{U}|$)) space.

2.2.1 Invertible Bloom Filter

We now describe the Invertible Bloom Filter (IBF), which can simultaneously calculate D_{A-B} and D_{B-A} using O(d) space. We start with some intuition. An IBF is named because it is similar to a standard Bloom Filter—except that it can, with the right settings, be *inverted* to yield some of the elements that were inserted.

Recall that in a counting Bloom Filter [25], when a key K is inserted, K is hashed into several locations of an array and a count, `count`, is incremented in each hashed location. Deletion of K is similar except that `count` is decremented. A check for whether K exists in the Filter returns yes (with high probability) if all locations that K hashes to have non-zero `count` values.

An IBF has another crucial field in each array location (cell) besides the count. This is the `idSum`: the sum of all key IDs that hash into that cell. Now imagine that

two peers, Peer 1 and Peer 2, doing set reconciliation on a large file of a million blocks independently compute an IBF B_1 and B_2 of size 100 for each block they contain. Note that if each block is hashed to 3 cells, an average of 30,000 block keys hash onto the same cell. Thus, each count will be large and the `idSum` in each cell will be the sum of a large number of IDs. Assume that Peer 1 sends B_1 to Peer 2 using a small amount of bandwidth to send around 200 counters. Peer 2 then proceeds to "subtract" its IBF B_2 from B_1. It does this cell by cell, by subtracting the `count` and the `idSum` in the corresponding cells of the two IBF's.

Intuitively, if Peer 1 and Peer 2 have almost the same blocks, except for say 25 different blocks among the total of 1 million blocks, all the common IDs that hash onto the same cell will be subtracted from `idSum`, leaving only the sum of the unique IDs (that belong to one peer and not the other) in the `idSum` of each cells. Further, if the set difference is small, we are throwing 25 balls randomly into say 3 out of 100 cell. We will prove that there is a high probability that at least one cell is "pure" in that it has at one unique element by itself.

A "pure" cell signals its "purity" by having its `count` field equal to 1, and, in that case, the `idSum` field yields the ID of one element in the set difference. We delete this element from all cells it has hashed to in the difference IBF by the appropriate subtractions; this, in turn, may free up more pure elements that it can in turn be decoded, to ultimately yield all the elements in the set difference.

The reader will quickly see subtleties. First, what if four IDs W, X, Y, Z satisfy $W + X = Y + Z$? To reduce the likelihood of decoding errors, IBF's use a third field in each cell as a checksum: the sum of the hashes of all IDs that hash into a cell, but using a different hash function than that used to determine the cell. Second, if Peer 2 has an element that Peer 1 does not, could the subtraction $B_1 - B_2$ produce negative values for `idSum` and `count`? Indeed, it can and the algorithm deals with this: for example, in `idSum` by using XOR instead of addition and subtraction, and in recognizing purity by `count` values of 1 *or* -1. While IBF's were introduced earlier [23, 28], whole IBF subtraction is new to this work; hence, negative counts did not arise in [23, 28].

Figure 2.1 summarizes the encoding of a set S and Figure 2.2 gives a small example of synchronizing the sets at Peer 1 (who has keys V, W, X and Y and Peer 2

(who has keys W, Y and Z. Each element is hashed into 3 locations: for example, X is hashed into buckets 1, 2 and 3. While X is by itself in bucket 3, after subtraction Z also enters, causing the `count` field to (wrongly) be zero. Fortunately, after subtraction, bucket 4 becomes pure as V is by itself. Note that bucket 5 is also pure with Z by itself, and is signaled by a count of -1. Decoding proceeds by first deleting either V or Z, and then iterating till no pure cells remain.

Encode First, assume that we have an oracle which, given S_A and S_B, returns the size of the set difference, d. We will describe the construction of such an oracle in Section 2.2.2. We allocate an IBF, which consists of a table B with $n = \alpha d$ cells, where $\alpha \geq 1$. Each cell of the table contains three fields (`idSum`, `hashSum` and `count`) all initialized to zero.

Additionally, hosts agree on two hash functions, H_c and H_k, that map elements in $\mathbb{U}$ uniformly into the space $[0, h)$, where $h \leq u$. Additionally, they agree on a value, k, called the `hash_count`. The algorithm for encoding a set S into an IBF is given in Algorithm 1 and illustrated in Figure 2.1. For each element in S, we generate k distinct random indices into B. To do this we recursively call $H_k()$ with an initial input of s_i and take the modulus by n until k distinct indices have been generated. Then for each index j returned, we XOR s_i into $B[j]$.`idSum`, XOR $H_c(s_i)$ into $B[j]$.`hashSum`, and increment $B[j]$.`count`.

Algorithm 1: IBF Encode

for $s_i \in S$ **do**
  for j in HashToDistinctIndices(s_i, k, n) **do**
    $B[j]$.`idSum` = $B[j]$.`idSum` $\oplus s_i$
    $B[j]$.`hashSum` = $B[j]$.`hashSum` $\oplus H_c(s_i)$
    $B[j]$.`count` = $B[j]$.`count` + 1

Subtract For each index i in two IBF's, B_1 and B_2, we subtract $B_2[i]$ from $B_1[i]$. Subtraction can be done in place by writing the resulting values back to B_1, or non-destructively by writing values to a new IBF of the same size. We present a non-destructive version of this algorithm in 2. Intuitively, this operation eliminates common

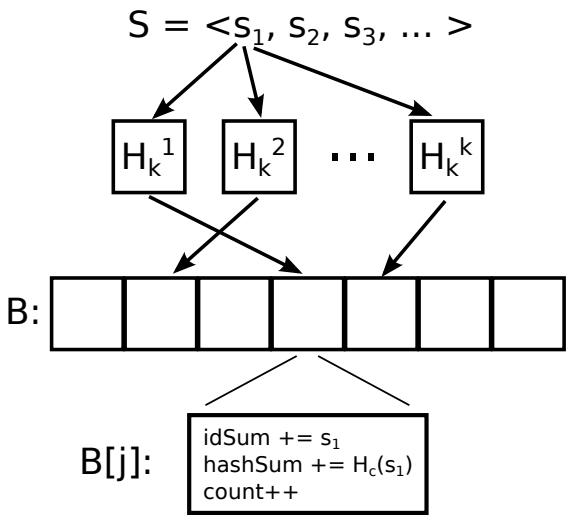


Figure 2.1: IBF Encode. Each element in set, S, is deterministically mapped to multiple cells of the IBF table, B. For each $B[j]$ an element is mapped to, the `idSum`, `hashSum`, and `count` of that cell are updated.

elements from the resulting IBF as they cancel from the `idSum` and `hashSum` fields as shown in Figure 2.2.

Algorithm 2: IBF Subtract ($B_3 = B_1 - B_2$)

for i in $0, \ldots, n-1$ **do**

$B_3[i]$.`idSum` $= B_1[i]$.`idSum` $\oplus B_2[i]$.`idSum`

$B_3[i]$.`hashSum` $= B_1[i]$.`hashSum` $\oplus B_2[i]$.`hashSum`

$B_3[i]$.`count` $= B_1[i]$.`count`- $B_2[i]$.`count`

Decode We have seen that to decode an IBF, we must recover "pure" cells from the IBF's table. Pure cells are those whose `idSum` matches the value of an element s in the set difference. In order to verify that a cell is pure, it must satisfy two conditions: the `count` field must be either 1 or -1, and the `hashSum` field must equal H_c(`idSum`). For example, if a cell is pure, then the sign of the `count` field is used to determine which set s is unique to. If the IBF is the result of subtracting the IBF for S_B from the IBF for S_A, then a positive `count` indicates $s \in D_{A-B}$, while a negative count indicates $s \in D_{B-A}$.

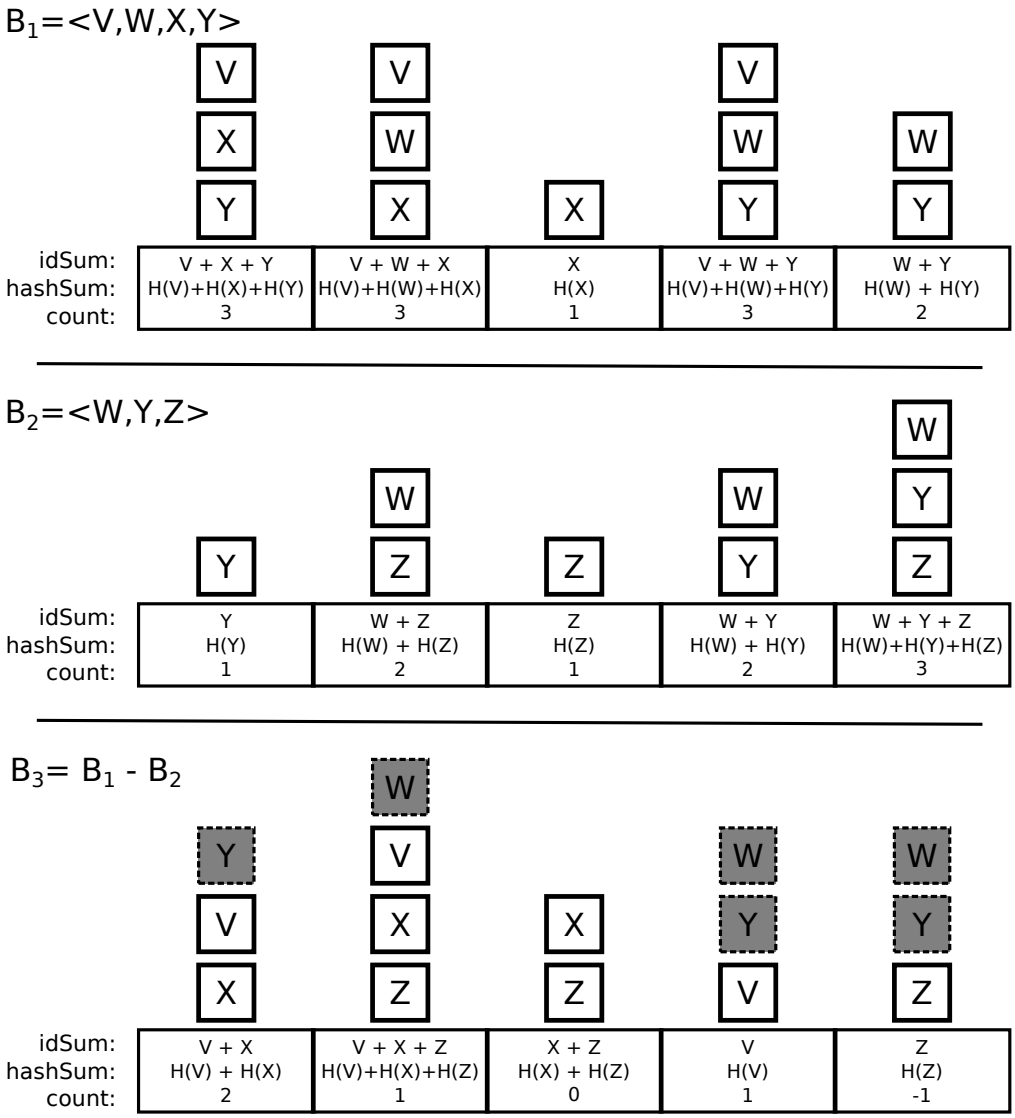


Figure 2.2: IBF Subtract. IBF B_3 is the result of subtracting IBF B_2 from IBF B_1. Each cell in B_3 is produced by subtracting the corresponding cells in B_1 and B_2. To subtract cells, the `idSum` and `hashSum` fields are XOR'ed, and `count` fields are subtracted. The elements common to B_1 and B_2 (shown shaded) are cancelled during the XOR operation.

Decoding begins by scanning the table and creating a list of all pure cells. For each pure cell in the list, we add the value s =`idSum` to the appropriate output set (D_{A-B} or D_{B-A}) and remove s from the table. The process of removal is similar to that of insertion. We compute the list of distinct indices where s is present, then decrement `count` and XOR the `idSum` and `hashSum` by s and $H_c(s)$, respectively. If any of these cells becomes pure after s is removed, we add its index to the list of pure cells.

Decoding continues until no indices remain in the list of pure cells. At this point, if all cells in the table have been cleared (i.e. all fields have value equal to zero), then the decoding process has successfully recovered all elements in the set difference. Otherwise, some number of elements remains encoded in the table, but insufficient information is available to recover them. The pseudocode is given in Algorithm 3 and illustrated in Figure 2.3.

2.2.2 Strata Estimator

In order to effectively utilize an IBF, we must determine the approximate size of the set difference, d since approximately $1.5d$ cells are required to successfully decode. We now give an algorithm for estimating d using O(log(u)) memory. We start, once more, with the intuition. If the set difference is large, we know that estimators such as random samples [31] and Min-wise Hashing [15, 16] will work well. However, we wish to design an estimator that can accurately estimate very small differences (say 10) even when the set sizes are large (say million).

Flajolet and Martin (FM) [26] give an elegant way to estimate set sizes (not differences). The FM estimator uses $\log u$ bits, where u is the size of the universe of set values. Each bit i in the estimator is the result of sampling the set with probability $1/2^i$; bit i is set to 1, if at least 1 element is sampled when sampling with this probability. Intuitively, if there are $2^4 = 16$ distinct values in the set, then when sampling with probability $1/16$, it is likely that bit 4 will be set. Thus the estimator returns 2^I as the set size, where I is the highest stratum (i.e., bit) such that bit I is set.

While FM data structures are useful in estimating the size of two sets, they do help in estimating the size of the difference as they contain no information that can be used to approximate which elements are common. However, we can *sample the set*

Algorithm 3: IBF Decode ($B \rightarrow D_{A-B}, D_{B-A}$)

for $i = 0$ to $n - 1$ **do**
  if $B[i]$ is pure **then**
    Add i to pureList
while pureList $\neq \emptyset$ **do**
  i=pureList.dequeue()
  if $B[i]$ is not pure **then**
    continue
  s=$B[i]$.idSum
  c=$B[i]$.count
  if $B[i]$.count > 0 **then**
    add s to D_{A-B}
  else
    add s to D_{B-A}
  for j in DistinctIndices(s, k, n) **do**
    $B[j]$.idSum = $B[j]$.idSum $\oplus s$
    $B[j]$.hashSum = $B[j]$.hashSum $\oplus H_c(s)$
    $B[j]$.count = $B[j]$.count- c
for $i = 0$ to $n - 1$ **do**
  if $B[i]$.idSum $\neq 0$ OR $B[i]$.hashSum $\neq 0$ $B[i]$.count $\neq 0$ **then**
    return FAIL
return SUCCESS

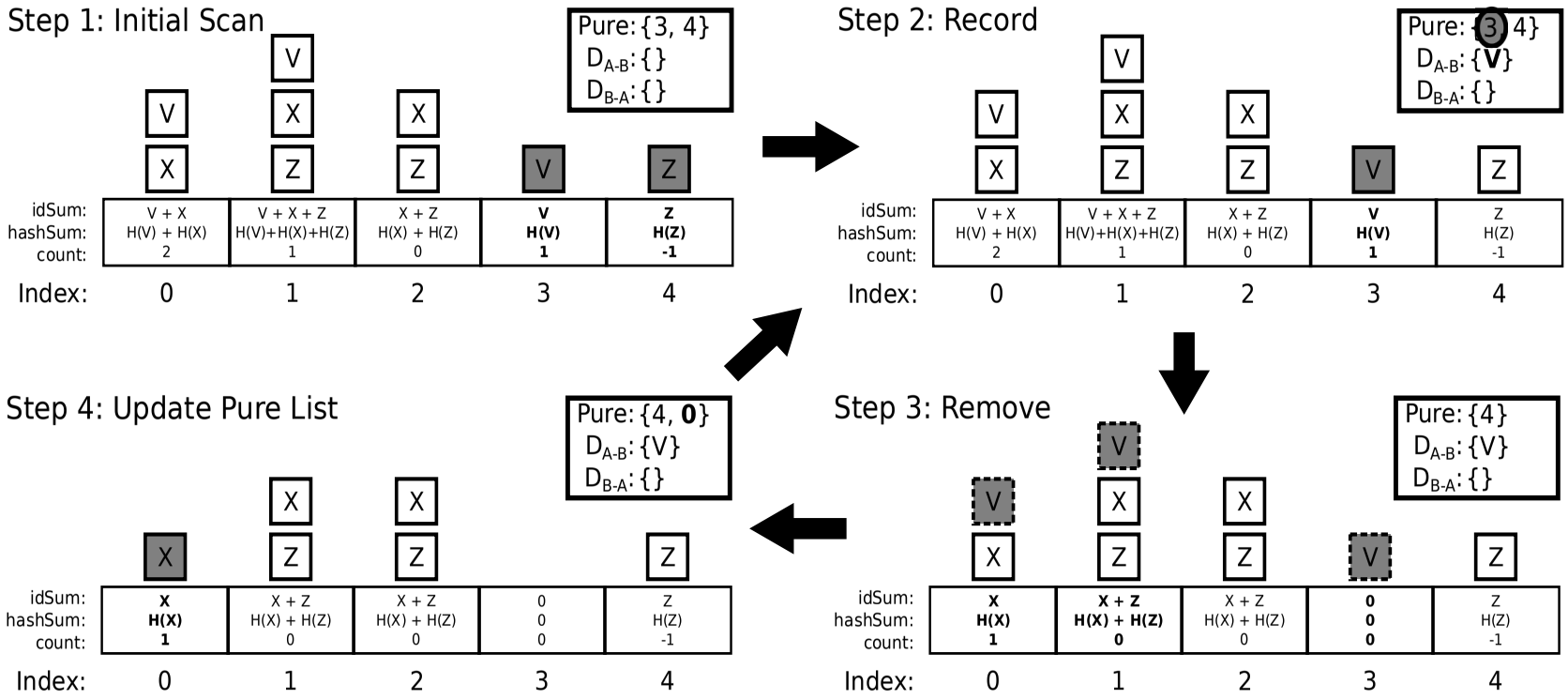


Figure 2.3: IBF Decode. We initially scan the IBF for pure cells, and add these indices (3&4) to the Pure list (Step 1). In Step 2, we dequeue the first index from the Pure list, verify that the cell is still pure, then add the value of the `idSum` as an element of the appropriate output set ($V \rightarrow D_{A-B}$). We then remove V from the IBF by computing the k distinct indices where it was inserted during encoding, decrementing the `count` of those cells, and XORing V and $H_c(V)$ from their `idSum` and `hashSum`, respectively. Finally, if any of these cells have now become pure, we add them to the Pure list (Step 4).

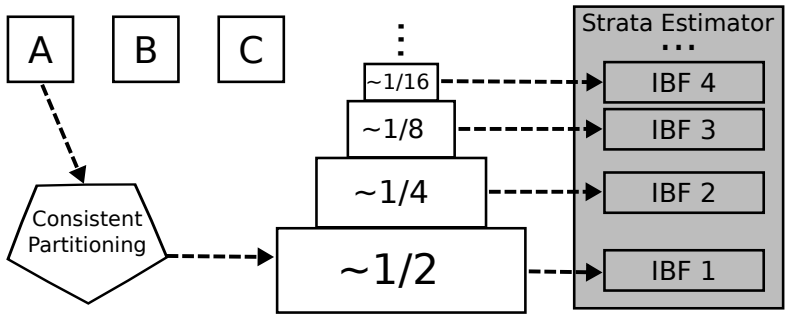


Figure 2.4: A set of elements is partitioned such that rough half are encoded in the IBF constituting the first stratum of the estimator, roughly a quarter in the IBF constituting the second stratum, and so on.

difference using the same technique as FM. Given that IBF's can compute set differences with small space, we use a hierarchy of IBF's as strata. Thus Peer A computes a logarithmic number of IBF's (strata), each of some small fixed size, say 80 cells.

Compared to the FM estimator for set sizes, this is very expensive. Using 32 strata of 80 cells is around 32 Kbytes but is the only estimator we know that is accurate at very small set differences and yet can handle set-difference sizes up to 2^{32}. In practice, we build a lower overhead composite estimator that eliminates higher strata and replaces them with a MinWise estimator, which is more accurate for large differences. Note also that 32 Kbytes is still inexpensive when compared to the overhead of naively sending a million keys.

Proceeding formally, we stratify $\mathbb{U}$ into $L =$log(u) partitions, $P_0, \ldots, P_L$, such that the range of the ith partition covers $1/2^{i+1}$ of $\mathbb{U}$. For a set, S, we encode the elements of S that fall into partition P_i into the ith IBF of the Strata Estimator. We show this visually in Figure 2.4. Partitioning $\mathbb{U}$ can be easily accomplished by assigning each element to the partition corresponding to the number of trailing zeros in its binary representation.

Each host then transmits the Strata Estimator for its set to its remote peer. For each IBF in the Strata Estimator, the host subtracts the corresponding remote IBF from the local IBF, then attempts to decode. If decoding the ith IBF is successful, then we estimate that the size of the set difference is the number of elements recovered scaled the by 2^{i+1}. We give the pseudocode in Algorithms 4 and 5, and illustrate the estimation

process in Figure 2.5.

In this description, we assumed that the elements in S_A and S_B were uniformly distributed throughout $\mathbb{U}$. If this condition does not hold, the partitions formed by counting the number of low-order zero bits may skew the size of our partitions such that stratum i does not hold roughly $|S|/2^{i+1}$. This can be easily solved by choosing some hash function, H_z, and inserting each element, s, into the IBF corresponding to the number of trailing zeros in $H_z(s)$.

Algorithm 4: Strata Estimator Encode

for $s \in S$ **do**
  $z =$ Number of trailing zeros in s.
  Insert s into the zth IBF

Algorithm 5: Strata Estimator Decode

for $i = 0$ to $\log(u)$ **do**
  if ($\text{IBF}_1[i]$ - $\text{IBF}_2[i]$) decodes **then**
    return $2^{(i+1)}\times$ element count in ($\text{IBF}_1[i]$ - $\text{IBF}_2[i]$)

2.3 Analysis

In this section we review and prove theoretical results concerning the efficiency of Invertible Bloom Filters and our stratified sampling scheme.

Theorem 1. *Let S and T be disjoint sets with d total elements, and let B be an invertible Bloom filter with $C = (k+1)d$ cells, where k is the number of random hash functions in B, and with at least $\Omega(k \log d)$ bits in each hashSum field. Suppose that (starting from a Bloom filter representing the empty set) each item in S is inserted into B, and each item in T is deleted from B. Then with probability at most $O(d^{-k})$ the IBFDecode operation will fail to correctly recover S and T.*

Proof. The proof follows immediately from the previous analysis for invertible Bloom filters (e.g., see [28]). □

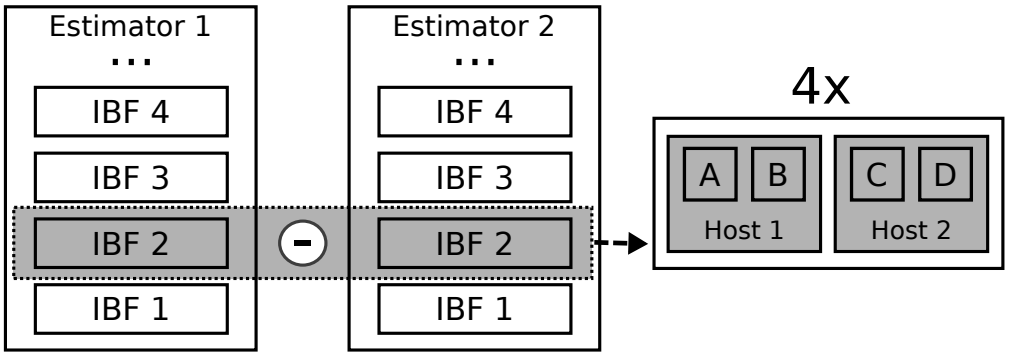


Figure 2.5: To estimate the size of the difference using a Strata Estimator, the strata for two estimators are evaluated pairwise, beginning with the strata containing the largest partition of the elements. The IBF subtract and decode operations are performed on each pair. If the ith pair sucessfully decodes, then the number of elements recovered is scaled by 2^i. In this example, the second pair of strata is the first to successfully decode and four elements are recovered. Hence, the estimate returned is $4 \times 2^2 = 16$.

We also have the following.

Corollary 1. *Let S and T be two sets having at most d elements in their symmetric difference, and let B_S and B_T be invertible Bloom filters, both with the parameters as stated in Theorem 1, with B_S representing the set S and B_T representing the set T. Then with probability at most $O(d^{-k})$ we will fail to recover S and T by applying the IBFSubtract operation to B_S and B_T and then applying the IBFDecode operation to the resulting invertible Bloom filter.*

Proof. Define $S' = S - T$ and $T' = T - S$. Then S' and T' are disjoint sets of total size at most d, as needed by Theorem 1. □

Let us next consider the accuracy of our stratified size estimator. Suppose that a set S has cardinality m, and let i be any natural number. Then the ith stratum of our Strata Estimator for S has expected cardinality $m/2^i$. The following lemma shows that our estimators will be close to their expectations with high probability.

Lemma 1. *For any $s > 1$, with probability $1-2^{-s}$, the cardinality of each of the stratum numbered up to i is within a multiplicative factor of $1 \pm O(\sqrt{s2^i/m})$ of its expectation.*

Proof. Let Y_j be the size of the jth stratum, for $j = 0, 1, \ldots, i$, and let μ_j be its expectation, that is, $\mu_j = E(Y_j) = m/2^j$. By standard Chernoff bounds (e.g., see [38, 39]), for $\delta > 0$,

$$\Pr\left(Y_j > (1+\delta)\mu_j\right) < e^{-\mu_j \delta^2/4}$$

and

$$\Pr\left(Y_j < (1-\delta)\mu_j\right) < e^{-\mu_j \delta^2/4}.$$

By taking $\delta = \sqrt{4(s+2)(2^i/m)\ln 2}$, which is $O(\sqrt{s2^i/m})$, we have that the probability that a particular Y_j is not within a multiplicative factor of $1 \pm \delta$ of its expectation is at most

$$2e^{-\mu_j \delta^2/4} \leq 2^{-(s+1)2^{i-j}}.$$

Thus, by a union bound, the probability that any Y_j is not within a multiplicative factor of $1 \pm \delta$ of its expectation is at most

$$\sum_{j=0}^{i} 2^{-(s+1)2^{i-j}},$$

which is at most 2^{-s}. □

Putting these results together, we have the following:

Theorem 2. *Let ϵ and δ be constants in the interval $(0, 1)$, and let S and T be two sets whose symmetric difference has cardinality d. If we encode the two sets with our Strata Estimator, in which each IBF in the estimator has C cells using k hash functions, where C and k are constants depending only on ϵ and δ, then, with probability at least $1 - \epsilon$, it is possible to estimate the size of the set difference within a factor of $1 \pm \delta$ of d.*

Proof. By the same reasoning as in Corollary 1, each IBF at level i in the estimator is a valid IBF for a sample of the symmetric difference of S and T in which each element is sampled with probability $1/2^{i+1}$. By Theorem 1, having each IBF be of $C = (k+1)g$ cells, where $k = \lceil \log 1/\epsilon \rceil$ and $g \geq 2$, then we can decode a set of g elements with probability at least $1 - \epsilon/2$.

We first consider the case when $d \leq c_0^2 \delta^{-2} \log(1/\epsilon)$, where c_0 is the constant in the big-oh of Lemma 1. That is, the size of the symmetric difference between S and T

is at most a constant depending only on ϵ and δ. In this case, if we take

$$g = \max\left\{2, \lceil c_0^2 \delta^{-2} \log(1/\epsilon) \rceil\right\}$$

and $k = \lceil \log 1/\epsilon \rceil$, then the level-0 IBF, with $C = (k+1)d$ cells, will decode its set with probability at least $1 - \epsilon/2$, as noted above. In this case, our estimator learns the exact set-theoretic difference between S and T, without error, with high probability.

Otherwise, let i be such that

$$d/2^i \approx c_0^2 \delta^2 / \log(1/\epsilon)$$

and let

$$g = \max\left\{2, \lceil c_0^2 \delta^{-2} \log(1/\epsilon) \rceil\right\}$$

and $k = \lceil \log 1/\epsilon \rceil$, as above. So, with probability $1-\epsilon/2$, using an IBF of $C = (k+1)d$ cells, we correctly decode the elements in the ith stratum, as noted above. By Lemma 1 (with $s = \lceil \log 1/\epsilon \rceil + 1$), with probability $1-\epsilon/2$, the cardinality of the number of items included in the ith stratum is within a multiplicative factor of $1 \pm \delta$ of its expectation, and all the levels below i are within a similar multiplicative factor of their expectations. Thus, with high probability, our estimate for d is within a $1 \pm \delta$ factor of d. □

2.4 The KeyDiff System

We now describe KeyDiff, a service that allows applications to leverage distributed set-difference calculation using Difference Digests. As shown in Figure 2.6, the KeyDiff service provides three operations, `add`, `remove`, and `diff`. Applications can add and remove keys from an instance of KeyDiff, then query the service to discover the set difference between any two instances of KeyDiff. These instances may be local to the host, or distributed across the network.

As an example, suppose a developer wants to write an efficient file synchronization application. In this case, the application running at each host would map files to unique keys and add these keys to a local instance of KeyDiff. In order to efficiently synchronize files, the application would then query KeyDiff for differences between the set stored locally and the set stored at a remote host. The KeyDiff service opens a TCP

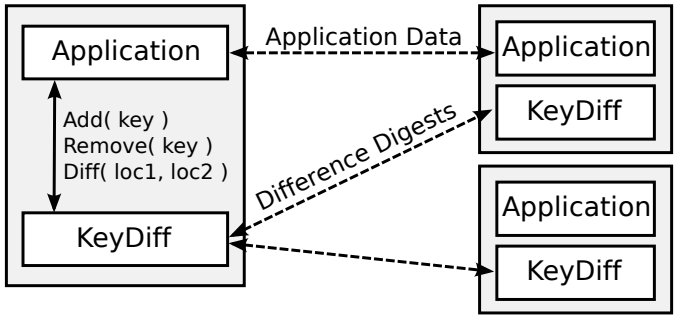


Figure 2.6: KeyDiff Service. The KeyDiff service can be run as part of an application or as a stand-alone server. Applications can add and remove keys from a KeyDiff instance. KeyDiff can compute the set difference between two instances and return this information to the application.

connection to the KeyDiff instance running on the remote host and runs the Difference Digest algorithm — first estimating the size of the difference by sending its Strata Estimator, and then computing the set difference after receiving an IBF of sufficient size from the remote host. Upon receiving the lists of unique keys from the KeyDiff service, the application can then perform the reverse mapping to identify and transfer the files that differ between the hosts.

The KeyDiff service is implemented using a client-server model and is accessed through an API written in C. All requests and responses between KeyDiff instances travel over a single TCP connection.

KeyDiff achieves efficient `diff` operations through the use of pre-computation. Internally, KeyDiff maintains an estimator structure, which can be quickly updated online as keys are added and removed. This structure is quickly serialized and transmitted in response to requests from other instances of KeyDiff. However, computing IBF's online is difficult since the number of cells required is dependent on the size of the difference, which is a value not known until after the estimation phase. Thus, in scenarios where computation is a bottleneck, KeyDiff can be configured to update several IBF's of pre-determined sizes online. Hence, the computational cost of building the IBF can be amortized across all of the calls to `add` and `remove`. This is reasonable because the cost of incrementally updating the Strata Estimator and a few IBF's when a key is added or removed is very small (a few microseconds) and should be a small fraction of

the time required for the application to create or store the object corresponding to the key.

For example, if the application is a peer-to-peer application and is synchronizing file blocks, the cost to store a new block on disk will be at least a few msec, and the small additional latency of microseconds to incrementally update a few IBF's will be inconsequential. We will show in the evaluation that if the IBF's are precomputed, then the latency of `diff` operations can be extremely small (100's of microseconds) for small set differences. In particular, this may help speed up the last phase of a P2P file download when only small set differences remain.

2.5 Evaluation

Our evaluation seeks to provide practical insights into the configuration and performance of Difference Digests and the KeyDiff system. To do so, we address the following four questions. First, what are the optimal parameters for an Invertible Bloom Filter? (Section 2.5.1). Second, how should one tune the Strata Estimator to balance accuracy and overhead? (Section 2.5.2). Third, how do IBF's compare with the existing Approximate Reconcilliation Tree technique? (Section 2.5.3). Finally, for what range of differences are Difference Digests most effectiveness in the entire KeyDiff system compared to using earlier approaches? (Section 2.5.4).

For our evaluation, we assume that $\mathbb{U}$ is all 32-bit values. Hence, we allocate 12 bytes for each IBF cell, with 4 bytes given to each `idSum`, `hashSum` and `count` field. We created pairs of files, each containing *keys* from $\mathbb{U}$. We exercised two degrees of freedom in creating pairs of files: the number of keys in each file, and the number of values that differed between the files, which we refer to as the files' *delta*. To create a pair of files, we randomly chose the specified number of keys from $\mathbb{U}$ and wrote them to the first file. The second file was created by copying the first file, then randomly selecting *delta* keys to delete. Thus, the second file is a subset of the first. For each experiment we created 100 file pairs. As our objective is set reconciliation, we consider an experiment successful only if we are able to successfully determine all of the elements in the set difference.

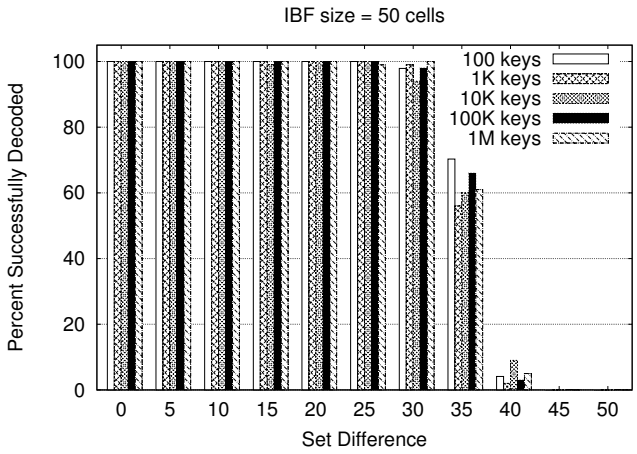


Figure 2.7: Rate of successful IBF decode shown for sets of various sizes. The ability to decode an IBF scales with the size of the set difference, not the set sizes. The IBF contained 50 cells and used 4 hash functions.

2.5.1 Tuning the IBF

We start by validating the property that IBF size scales with the size of the set difference not the total set size. To do so, we generated sets with 100, 1K, 10K, 100K and 1M keys, and deltas between 0 and 50. We then compute the set difference using an IBF with 50 cells.

Figure 2.7 shows the success rate for recovering all of the keys in the set difference. We see that for deltas up to 25, the IBF decodes completely with extremely high probability, regardless of set size. Between set differences of 30 and 40, the rate of successful decoding declines for all set sizes. This occurs as the number of collisions in the IBF prevents an adequate number of pure cells from being uncovered. We also see that at deltas of 45 and 50 the collision rate is so high that no IBF's are able to completely decode. The results in Figure 2.7 confirm that the probability of decoding success is independent of the original set sizes.

Determining table size and number of hash functions Both the size of the table and the `hash_count` are critical in determining the number of collisions and, hence, the rate of successful decoding. We now investigate the effects of these parameters. To evaluate the effect of `hash_count`, we attempted to decode sets with 100 keys

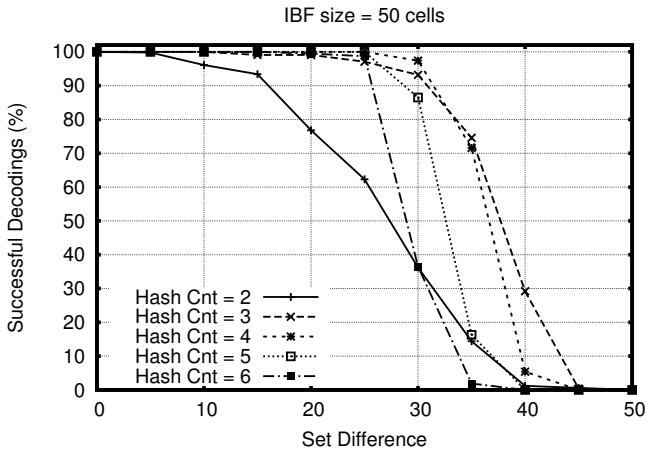


Figure 2.8: Probability of successful decoding for IBF's with varying `hash_count`'s. Each experiment was run with 100-element sets and replicated 1000 times.

and deltas between 0 and 50 using an IBF with 50 cells. As we have seen above, the size of the sets does not influence decoding success rates. Hence, these results are representative for arbitrarily large sets. We ran this configuration for 1000 file pairs using `hash_count`'s between 2 and 6. Our results are displayed in Figure 2.8.

For deltas less than 30, `hash_count` = 4 decodes 100% of the time, while higher and lower values for `hash_count` show degraded success rates. Intuitively, lower hash counts do not provide equivalent decode rates since each pure cell processed only removes the key from a small number of other cells. Higher values of `hash_count` avoids this problem but a larger value for `hash_count` also decreases the probability there will initially be a cell in the table that is pure. For deltas greater than 30, `hash_count` = 3 provides the highest rate of successful decoding. However, at smaller deltas, `hash_count` = 3 is less reliable than `hash_count` = 4, with approximately 98% success for deltas from 15 to 25 and 92% at 30.

Given that failed decodings require expensive retransmission, we minimize the probability of decoding failure as follows. Figure 2.9 quantifies the combination of IBF sizes and `hash_count`'s necessary to recover set differences with 99% certainty. We used 100 element sets with deltas ranging from 0 to 100 and attempted to recover the set difference with IBF's containing up to 150 cells. We replicated this experiment 1000 times per IBF size. Each curve in Figure 2.9 plots the lowest number of cells

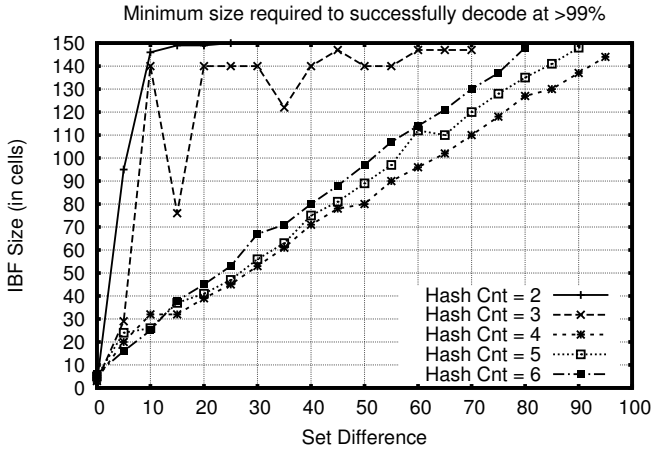


Figure 2.9: The minimum number of IBF cells required to ensure that the set difference can be completely recovered with 99% certainty. We see that `hash_count` equal to 4 requires the least memory for deltas less than 100.

above which 99% or more of the experiments successfully decoded. We see that a `hash_count` of 4 achieves full recovery in 99% of the experiments with the smallest memory footprint for all deltas greater than 15.

To understand the memory overhead (ratio of IBF's cells to set-difference size), we used sets contain 100K elements and varied the number of cells in each IBF as a proportion of the delta. We then evaluated the memory overhead required for 99% of the replicates to completely decode a wide range of deltas. We plot this data in Figure 2.10. Deltas below 100 all require at least of 50% overhead to completely decode. However, beyond set differences of 1000, the memory overhead reaches an asymptote as the size of the set difference continues to grow. As before, we see a `hash_count` of 4 decodes consistently with less than 5 or 6, but interestingly, `hash_count` = 3 has the lowest memory overhead at all deltas greater than 400.

2.5.2 Tuning the Strata Estimator

Since the Strata Estimator is constructed from several IBF's, we first look at which IBF size and `hash_count` provides the most accurate estimation. Based on our findings from Section 2.5.1, we know that `hash_count` = 4 has the most consistent

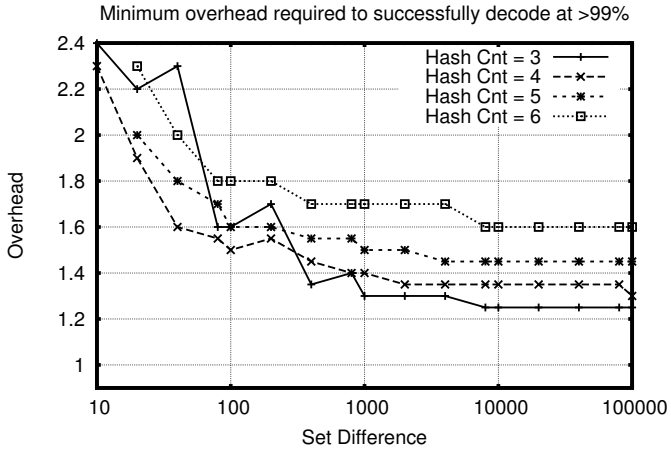
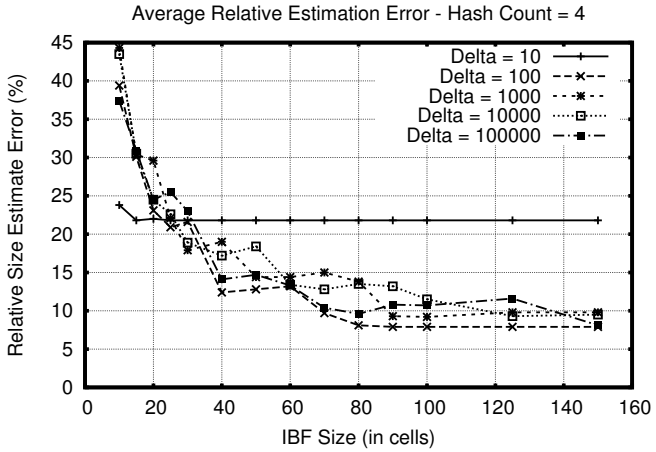


Figure 2.10: We evaluate 100K sets with a wide range of delta and plot the minimum space overhead (IBF cells/delta) required to ensure that the set difference can be completely recovered with 99% certainty.

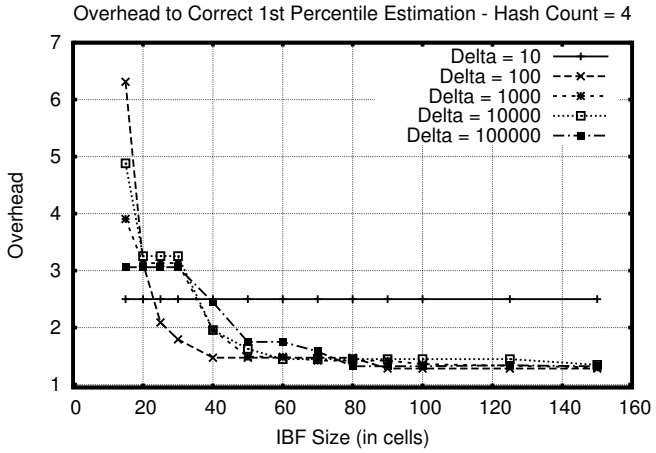
odds of successful recovery at IBF sizes less than 400 cells. Hence, we focus on Strata Estimators whose IBF's use a `hash_count` of 4.

In Figure 2.11 we plot the performance of Strata Estimators with varying IBF sizes run on sets containing 100K elements and a wide range of deltas. Figure 2.11a plots the average relative error between the value returned by the Strata Estimator and the true size of the set difference. We see that the relative error drops sharply as the number of cells per IBF is increased from 10 to 40. The rate of improvement slows as more cells are added and flattens after 100 cells. Hence, choosing a Strata Estimator with IBF's containing between 80 and 100 cells gives the best trade off between memory overhead and estimation error.

Recall that if the Strata Estimator over-estimates, the subsequent IBF will be unnecessarily large and waste bandwidth. However, if the Strata Estimator under-estimates, then the subsequent IBF may not decode and cost an expensive retransmission. To mitigate the affects of under-estimation, the values returned by the estimator should be scaled by some factor. For our evaluation, we report the scaling overhead required such that 99% of the estimates will be greater than or equal to the true difference. Figure 2.11b plots the scaling overhead required for the 1st percentile of estimations. Again we see that Strata Estimators containing large IBF's reach a point of diminishing



(a) Average Relative Error - Hash Count = 4



(b) 1st Percentile Estimation Correction - Hash Count = 4

Figure 2.11: Strata Estimator performance versus strata size. Estimator accuracy improves with larger strata up to 100 cells. To mitigate the affects of under-estimation, Figure 2.11b plots the scaling overhead necessary to ensure that 99% of the estimates are greater than or equal to the true delta's size.

returns with a good size-to-accuracy trade off occurring around 80 cells per IBF. With 80 cells per IBF, any estimate returned by a Strata Estimator should be scaled by a factor of 1.47.

However, deltas less than 100 are an exception to these trends. As seen in Figure 2.11, deltas of 10 do not respond to an increase in memory, reporting a relative error of 22% and scaling overheads for 1st percentile estimates of 2.5. Intuitively, we expect 50% of the elements in the difference to accumulate in the first stratum's IBF. Therefore, any variance in the estimated value is dictated by the number of elements hashed into the first stratum, and not the size of the IBF.

Strata Estimator vs. Min-wise We next compare our Strata Estimator to the Min-wise Estimator (see Section 2.1). For our comparison we used sets with 100K elements and deltas ranging from 10 to 100K. Given knowledge of the approximate total set sizes a priori, the number of strata in the Strata Estimator can be adjusted to conserve communication costs. From the data in Figure 2.9, we see that an IBF with 80 cells can successfully decode deltas up to 50 with 99% probability. Therefore, a Strata Estimator with 12 strata containing 80 cells per IBF has a maximum range of $50 \times 2^{12} = 200K$. At 12 bytes per IBF cells, this configuration requires approximately 11.5 KB of space. Alternatively, one could allocate a Min-wise estimator with 2880 4-byte hashes in the same space.

In Figure 2.12 we compare a Strata Estimator and a Min-wise estimator each consuming equal memory footprints. In Figure 2.12a, we see that the Min-wise estimator has diminishing errors ranging from 15.2% to 0% as deltas increase beyond 1000. The Strata Estimator returns values with average relative errors consistently between 15.6% and 9.6% for the same range. However, the accuracy of the Min-wise estimator deteriorates rapidly for smaller delta values, reaching 149% at a delta of 10, compared to 21.8% for the Strata Estimator. This is not surprising because as the size of the difference shrinks relative to the total size of the sets, the odds that hashes created by elements in the difference are included in the array drops.

Figure 2.12b shows the scaling factors required such that 99% of the estimates from the Strata and Min-wise estimators are greater than or equal to the true delta. We

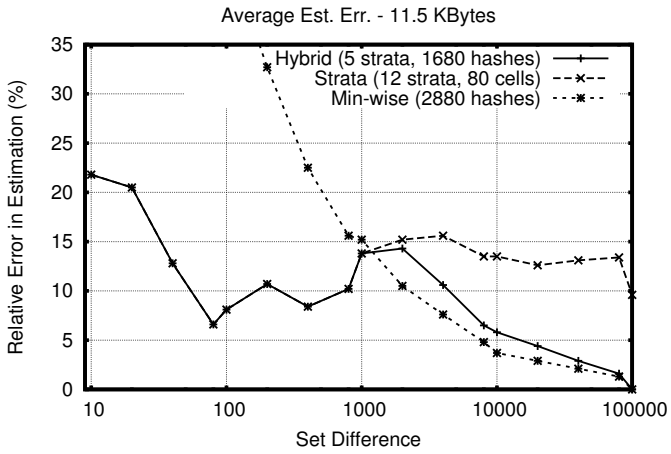
see that the overhead required by Min-wise increases significantly as the size of the set difference decreases. In fact, for all deltas below 400, the 1st percentile of Min-wise estimates are 0, resulting in infinite overhead. In contrast, we see that the Strata Estimator provides estimates for all delta values, and our second phase IBF will have sufficient space 99% of the time by scaling any estimation greater than 10 by 1.84.

Hybrid Estimator The Strata Estimator outperforms Min-wise for small differences, while the opposite occurs for large differences. This suggests the creation of a hybrid estimator that keeps the lower strata to ensure accurate small deltas estimation, while augmenting more selective strata with a Min-wise estimator.

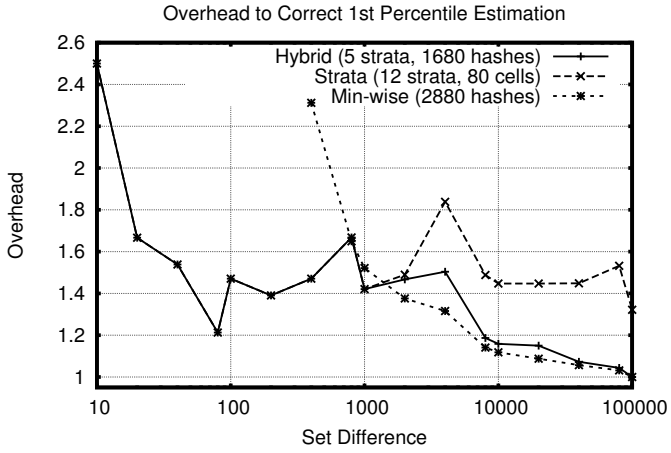
For our previous Strata Estimator configuration of 80 cells per IBF, each stratum consumes 960 bytes. Therefore, we can trade a stratum for 240 additional Min-wise hashes. From Figure 2.12 we see that the Strata Estimator performs better for deltas less than 1000. Therefore, we retain the lower 5 strata, giving the Strata Estimator component a range of $50 \times 2^5 = 1600$. With the remaining 6720 bytes, we allocate a Min-wise estimator with 1680 cells to answer any queries beyond the range of the lower strata.

Results from our Hybrid Estimator are plotted in Figure 2.12 against the results of the original Strata and Min-wise estimators. We see that the Hybrid Estimator mirrors the results of the Strata Estimator for all deltas up to 1000, as desired.

Difference Digest Configuration Guideline. By using the Hybrid Estimator in the first phase, we achieve an estimate greater than or equal to the true difference size 99% of the time by scaling the result by 1.67. In the second phase, we further scale by 1.25 to 2.3 and set `hash_count` to either 3 or 4 depending on the estimate from phase one. In practice, a simple rule of thumb is to construct an IBF in Phase 2 with twice the number of cells as the estimated difference. For estimates greater than 200, 3 hashes should be used and 4 hashes otherwise. At 12 bytes per IBF cell, we can reconcile the two sets at a cost of 25 to 46 bytes per element in the set difference.



(a) Average Relative Error



(b) 1st Percentile Estimation Correction

Figure 2.12: Comparison of Estimators. Average relative error and scaling overhead to correct down to the 1st percentile of estimates are shown when memory is constrained to 11.5KB. The Strata Estimator is configured with 12 strata each containing 80 cells. The Hybrid Estimator uses 5 IBF's each containing 80 cells and 1680 min-wise hashes.

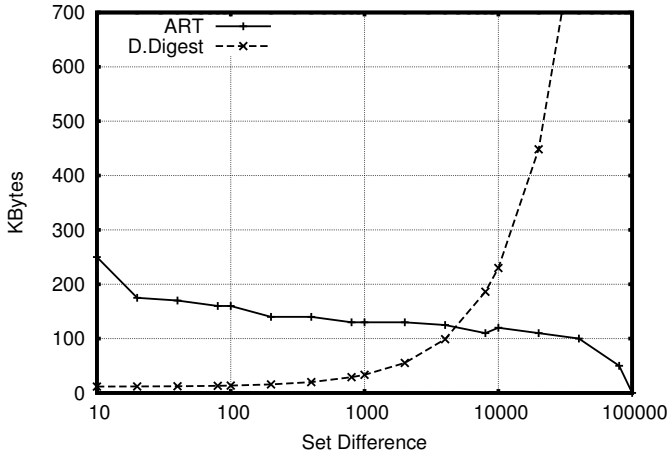


Figure 2.13: Data transmission required by ART and Difference Digests to recover 95% and 100% of the set difference, respectively, with 99% reliability.

2.5.3 Difference Digest vs. ARTree

Having understood how to configure a Difference Digest, we compare the Difference Digest to Approximate Reconciliation Trees (ART) [18]. We note that ART's were originally designed to compute *most but not all* the keys in $S_A - S_B$. Bloom Filters in ART lead to the possibility of false positives and the inadvertent pruning of branches that contain unique keys. To address this, the system built in [18] used erasure coding techniques to ensure that hosts received pertinent data. However, while this approach is reasonable for some P2P applications it may not be applicable to or desirable for all applications described at the beginning of this chapter.

Given that ART's were not designed for computing the complete set difference, we arbitrarily choose the standard of 95% of the difference 99% of the time and plot the amount of data required to achieve this level of performance with ART. For comparison, we also plot the data used by Difference Digest for both estimation and reconciliation to compute the complete difference 99% of the time. These results are shown in Figure 2.13.

The results show that the bandwidth required by ART decreases as the size of the difference increases. This intuitively makes sense since the compacted ART encodes the contents of S_B, which is diminishing in size as the size of the difference grows

($|S_B| = |S_A| - |D|$). Thus, the compacted ART requires fewer bits in its Bloom Filters to capture the nodes in B's ART. At the same time, while the size of the Estimator stays constant, the IBF grows at a rate of 24 Bytes (three 4-byte values and a factor of 2 inflation for accurate decoding) per key in the difference.

Algorithm Selection Guidelines. The results show that for small differences Difference Digests require an order of magnitude less bandwidth (175 to 14 KB) and are better up to a difference of 4,000 (4%). However, the fact that ART does better after 4K is misleading because we have allowed the ART to decode only 95% of the difference, which may be unacceptable in many applications.

2.5.4 KeyDiff Performance

Having configured the Difference Digest and seen its potential benefit over ART using *microbenchmarks*, we now examine the benefits of Difference Digest in system-level benchmarks using KeyDiff service described in Section 2.4. To do so, we quantify the performance of KeyDiff using Difference Digests versus ART's and the simplest approach of all, trading a complete list of keys (List). For these experiments, we deployed KeyDiff on two dual-processor quad-core Xeon servers running Linux 2.6.32.8. Our test machines were connected via 10 Gbps Ethernet and report an average RTT of 93μs.

Costs and Benefits of Incremental Updates For each algorithm, KeyDiff maintains a set of structures. Some of these structures, such as the Difference Digest estimators, can be updated online when a key is added or removed, while others, such as the IBF, are based on runtime parameters and must be generated on the fly. While adding elements incurs very low overhead, on-the-fly generation of an IBF or compacting an ART can introduce significant overhead to the diff operation. To address this issue, we investigate the costs and benefits of precomputing techniques.

List. For the basic List algorithm, new keys are added to the tail of a linked list and we transfer an entire listing of keys from host B to host A. Host A sorts both lists and scans them to identify unique keys. By storing the lists in stored order the diff operation can avoid the sorting phase during runtime at the cost of a slightly longer

sorted insertion.

ART. For each addition, a new interior and leaf node are added to the ART and all ancestor nodes must be updated. When a summary is requested, the tree is traversed, with interior nodes being added to one Bloom filter and the leaf nodes added to another. Since the value of interior nodes may change with each insertion, one would need to re-compute the Bloom filter for the interior nodes for each addition. However, the limitation can be removed through incrementally updating the compacted ART, using a counting bloom filter [25] and omitting the counters when the bloom filter is sent over the network.

Difference Digest. The estimators for the Difference Digest are maintained online as elements are added and removed from KeyDiff, however the IBF cannot be constructed until after the approximate size of the difference is known. We can avoid this runtime overhead by maintaining several IBF's of predetermined sizes with KeyDiff. Each key is added to all IBF's and the smallest IBF that should decode the estimated difference is returned. The number of IBF's to maintain in parallel will depend on the computing and bandwidth overheads encountered for each application.

In Table 2.1, we report the average time required to add a key to our various structures, both with and without precomputation during each add. We also report the duration that the KeyDiff server must spend computing before a response can be sent. We note that these times correspond closely with the number of memory locations each algorithm touches, with Min-wise computing and checking values for all of its hashes, `ART` traversing a path through its tree, and IBF updating `hash_count` cells. For most applications, the small cost (10's of microseconds) for incremental updates during each add operation should not dramatically affect overall application performance, while the speedup during `diff` (seconds) is a clear advantage.

`Diff` **Performance** We now look at time required to run the `diff` operation. As we are primarily concerned with the performance improvement as seen by an application built on top of these algorithms, we use incremental updates and measure the wall clock time required to compute the set difference.

For `ART` [18], we used 8 bits of storage per key with 70% allocated to storing

Table 2.1: Time required to add a key to KeyDiff and the time required to generate a KeyDiff response for various summary and estimation algorithms. The time per `add` call is averaged across the insertion of 1M keys. We see that precomputation increases each addition, but greatly reduces the time to generate a response.

Algorithm	Add (μs)	Serv. Compute
`List`	0.215	6.542 msec
`List` (sorted)	1.309	6.545 msec
`ART`	1.995	6,937.831 msec
`IBF`	N/A	3,957.847 msec
`IBF` (8x precompute)	31.320	0.022 msec
Min-wise (2880 hashes)	21.909	0.022 msec
Strata (12x80 cells)	4.303	0.021 msec
Hybrid (5x80 cells + 1680 hash)	17.139	0.023 msec

the ART's leaf nodes. In addition, when comparing the local ART to the compacted representation from the remote host, we allowed for 5 consecutive hits to occur in the bloom filter containing the internal nodes of the remote ART before pruning the path in the local tree. Based on the evaluation from [18], these parameters should recover 92% of the elements unique to the local host and be computationally efficient for small differences. Difference Digests are run with 11.5 KB dedicated to the Estimator, and 8 parallel, precomputed IBF's with sizes ranging from 256 to 400K cells in factors of 4.

We populated the first host, A with a set of unique 32-bit keys, S_A and copied a random subset of those keys, S_B to the other host, B. From host A, we then query KeyDiff to compute the set of unique keys. A KeyDiff instance at A can compute the keys unique to both hosts (S_{A-B} and S_{B-A}) with either Difference Digest or `List`, but `ART` only allows host A to compute S_{A-B}. To allow a fair comparison with `ART`, we limit our experiments to only consider the case where $S_B \subseteq S_A$ (Difference Digests or `List` do not need this assumption).

Our results can be seen in Figure 2.14a. We note that there are 3 components contributing to the latency for all of these methods, the time to generate the response at host B, the time to transmit the response, and the time to compare the response to the set stored at host A. Since we maintain each data structure online, the time for host B to generate a response is negligible and does not affect the overall latency.

We see from these results that the `List` shows fast performance across difference sizes, but performs particularly well as the size of the difference increases. Since the size of the data sent *decreases* as the size of the difference increase, the transmission time and the time to sort and compare at A decrease accordingly.

On the other hand, the Difference Digest performs best at small set differences and its latency increases linearly with the size of the difference. This agrees with our theoretical result that both the transmission size and the decoding time for an IBF is $O(D)$ words. Since the estimator is maintained online, the estimation phase conclude very quickly, often taking less than 1 millisecond. We also see that precomputing IBF's at various sizes is essential and significantly reduces the latency by eliminating the $O(|S_B|)$ computation at host B at runtime.

Interestingly, we observe that `ART` shows the lowest latencies when the set difference is very small or very large, with the longest latencies occurring when the difference is 10% of the total set size. When the difference is very small, S_B is close to the size of S_A and the time required to produce the compacted ART at host B dominates the latency. In the case that the difference is large, $|S_B|$ is small, hence computing the compact ART finishes quickly. Since there are many differences, A must traverse almost its entire ART, an operation that dominates the computation time.

In considering the resources required to for each algorithm, we see that `List` requires $4|S_B|$ bytes in transmission and touches $|S_A|+|S_B|$ values in memory. Difference Digest has a constant estimation phase followed by $6|D|$ to $24|D|$ bytes in transmission and $3|D| \times$ `hash_count` memory operations (3 values in each IBF cell). Finally, `ART` requires $|S_B|$ bytes of communication and roughly $|D|log(|S_A|)\times$ the hash count of the Bloom Filters in the compacted ART.

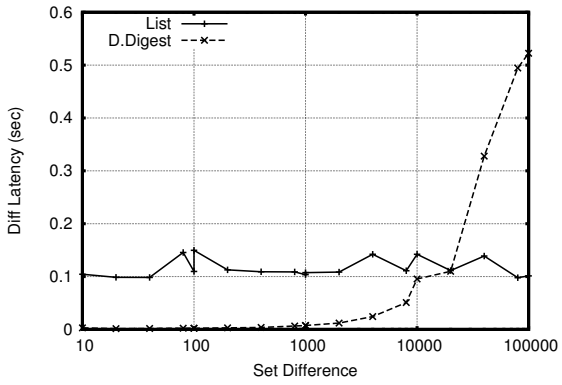
If our experimental setup were completely computation constrained, we would expect `List` to have superior perform at large differences and Difference Digest to shine at small difference. If we assume a `hash_count` of 4, then Difference Digest's latency is $12|D|$, while `List`'s is $|S_A| + |S_B| = 2|S_A| - |D|$. Thus, they will have equivalent latency at $|D| = \frac{2}{12-1}S_A$, or a difference of 18%. We note that `ART`'s runtime also scales with D, but for any practical set size, the $log(|S_A|)$ times the number of Bloom Filter hashes will be larger than IBF's constant factor of 12.

Guidance for Constrained Computation. We conclude that, for our test environment, precomputed Difference Digests are superior for small difference sizes (less than 10%), while sending a full list is preferable at difference sizes larger than 10%. In contrast, `ART`'s perform significantly slower than these other approaches. We argue that in environments where computation is severely constrained relative to bandwidth, this crossover point can reach up to 18%. In such scenarios, precomputation is vital to optimize `diff` performance.

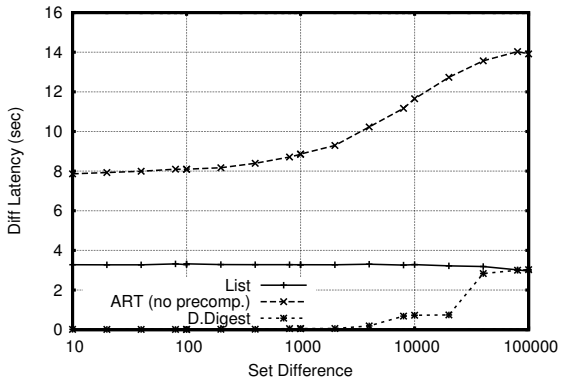
Varying Bandwidth We now investigate how the latency of each algorithm changes as the speed of the network decreases. For this we consider bandwidth of 10 Mbps and 100Kbps, which are speeds typical in wide-area networks and mobile devices, respectively. By scaling the transmission times from our previous experiments, we are able to predict the performance at slower network speeds. We show these results in Figure 2.14b and Figure 2.14c.

As discussed previously, the data required by `List`, `ART`, and Difference Digests is $4|S_B|$, $|S_B|$, and roughly $6|D|$, respectively. Thus, as the network slows and dominates the running time, we expect that `ART` will outperform `List` by a factor of 4 for all difference sizes. Since $|S_B| = |S_A| - |D|$, we expect the running times of Difference Digest and `ART` to be equal at $|D| = \frac{|S_A|}{6-1}$, or a difference of 20%. As the communication overhead for Difference Digest grows due to widely spaced precomputed IBF's the trade off point will move toward differences that are a smaller percentage of the total set size. However, for small to moderate set sizes and highly constrained bandwidths KeyDiff should create appropriately sized IBF's on demand to reduce memory overhead and minimize transmission time.

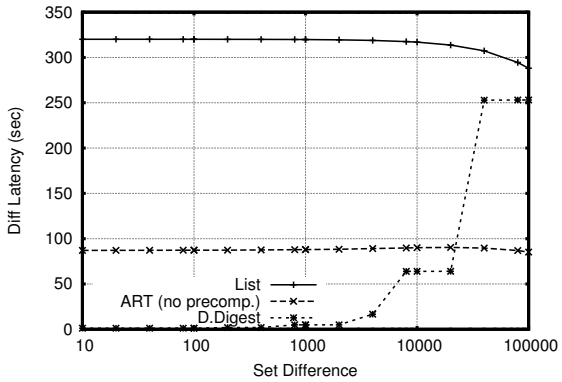
Guidance for Constrained Bandwidth. As bandwidth becomes a predominant factor in reconciling a set difference, the algorithm with the lowest data overhead should be employed. Thus, `ART` will have superior performance for differences greater than 20%. For smaller difference sizes, IBF will achieve faster performance, but the crossover point will depend on the size increase between precomputed IBF's. For constrained networks and moderate set sizes, IBF's could be computed on-demand to minimize time.



(a) Measured - 1.2 Gbps



(b) Modeled - 10 Mbps



(c) Modeled - 100 Kbps

Figure 2.14: Time to run KeyDiff `diff` for $|S_A|$= 1M keys and varying difference sizes. We show our measured results in 2.14a, then extrapolate the latencies for more constrained network conditions in 2.14b and 2.14c.

2.6 Conclusions

We have shown how Difference Digests can efficiently compute the set difference of data objects on different hosts using computation and communication proportional to the size of the set difference. The two main new ideas are whole set differencing for IBF's, and a new estimator that accurately estimates small set differences via a hierarchy of sampled IBF's.

We implemented Difference Digests in a KeyDiff service run on top of TCP that can be utilized by different applications such as Peer-to-Peer file transfer. There are three calls in the API (Figure 2.6): calls to add and delete a key, and a call to find the difference between a set of keys at another host.

Our evaluation found that IBF's and Strata Estimators have the lowest overhead and best accuracy when hashing each key to 4 cells. In addition, using 80 cells per stratum in the estimator worked well, and after the first 5 strata, Min-wise hashing provides better accuracy. Combined as a Difference Digest, the IBF and Estimator provide the best performance for differences less than 10 to 20% of the set size. This threshold changes with the ratio of bandwidth to computation, but could be estimated by observing the throughput during the estimation phase to choose the optimal algorithm.

While we have implemented a generic Key Difference service using Difference Digests, a next step is to use a KeyDiff service in some application to improve overall user-perceived performance. While we hope that KeyDiff can improve the endgame of a Peer-to-Peer protocol, we do not know if there are clear regimes in which it will improve file download times. Despite this, we hope the simplicity and elegance of Difference Digests and their application to the classical problem of set difference will inspire readers to more imaginative uses.

Chapter 2, in part, is a reprint of the material as it will appear in the article "Set Synchronization without Prior Context using Difference Digests" in the Proceedings of the Special Interest Group on Data Communications (SIGCOMM), Toronto, ON, Canada, August 2011. David Eppstein, Michael T. Goodrich, Frank Uyeda, George Varghese.

Chapter 3

Efficiently Measuring Bandwidth at All Time Scales

How can a manager of a computing resource detect bursts in resource usage that cause performance degradation without keeping a complete log? The problem is one of extracting a needle from a haystack; the problem gets worse as the needle gets smaller (as finer-grain bursts cause drops in performance) and the haystack gets bigger (as the rate of consumption increases). While this chapter addresses this general problem, we focus on detecting bursts of bandwidth usage, a problem that has received much attention [12, 42, 48] in modern data centers.

Data-center traffic consists of both large bandwidth-intensive flows and short latency-critical flows [33, 29]. While buffering in the network serves to smooth small bursts of traffic and maintain high link utilization, correlated bursts can fill switch buffers causing increases in latency and packet loss. Thus, network administrators would like to information about the timing, magnitude, and correlation between bursts to aid in debugging and tuning their distributed applications. To further complicate measurement efforts, the confluence of multi-Gigabit link speeds and small switch buffers have led to "microbursts", very short-lived events, which cause packet drops and large increases in latency.

The simplest definition of a *microburst* is the transmission of more than B bytes of data in a time interval t on a single link, where t is in the order of 100's of microseconds. For input and output links of the same speed, bursts must occur on several links

at the same time to overrun a switch buffer, as in the Incast problem [19, 42]. Thus, a more useful definition is the sending of more than B bytes in time t over *several* input links that are destined to the same output switch port. This general definition requires detecting bursts that are correlated in time across several input links.

Microbursts cause problems because data center link speeds have moved to 10 Gbps while commodity switch buffers use comparatively small amounts of memory (Mbytes). Since high-speed buffer memory contributes significantly to switch cost, commodity switches continue to provision shallow buffers, which are vulnerable to overflowing and dropping packets. Dropped packets lead to TCP retransmissions, which can cause millisecond latency increases that are unacceptable in data centers.

Administrators of financial trading data centers, for instance, are concerned with the microburst phenomena [7] because even a latency advantage of 1 millisecond over the competition may translate to profit differentials of $100 million per year [36]. While financial networks are a niche application, high-performance computing is not. Expensive, special-purpose switching equipment used in high-performance computing (e.g. Infiniband and FiberChannel) is being replaced by commodity Ethernet switches. In order for Ethernet networks to compete, managers need to identify and address the fine-grained variations in latencies and losses caused by microbursts. At the core of this problem is the need to identify the bandwidth patterns and corresponding applications causing these latency spikes so that corrective action can be taken.

Efficient and effective monitoring becomes increasingly difficult as faster links allow very short-lived phenomenon to overwhelm buffers. For a commodity 24-port 10 Gbps switch with 4 MB of shared buffer, the buffer can be filled (assuming no draining) in 3.2 msec by a single link. However, given that bursts are often correlated across several links and buffers must be shared, the time scales at which interesting bursts occur can be ten times smaller, down to 100's of μs. Instead of 3.2 msec, the buffer can overflow in 320 μs if 10 input ports each receive 0.4 MB in parallel. Assume that the strategy to identify correlated bursts across links is to first identify bursts on single links and then to observe that they are correlated in time. The single link problem is then to efficiently identify periods of length t where more than B bytes of data occur. Currently, t can vary from hundreds of microseconds to milliseconds and B can vary from 100's of

Kbytes to a few Mbytes. Solving this problem *efficiently* using minimal CPU processing and logging bandwidth is one of the main concerns of this chapter.

Although identifying "bursts" on a single link for a range of possible time scales and byte thresholds is challenging, the ideal solution should do two more things. First, the solution should efficiently extract flows responsible for such bursts so that a manger can reschedule or rate limit them. Second, the tool should allow a manager to detect bursts correlated in time across links. While the first problem can be solved using heavy-hitter techniques [40], we briefly describe some new ideas for this problem in our context. The second problem can be solved by archiving bandwidth measurement records indexed by link and time to a relational database, which can then be queried for persistent patterns. This requires an efficient summarization technique so that the archival storage required by the database is manageable.

Generalizing to Bandwidth Queries: Beyond identifying microbursts, we believe that modeling traffic at fine time scales is of fundamental importance. Such modeling could form the basis for provisioning NIC and switch buffers, and for load balancing and traffic engineering at fine time scales. While powerful, coarse-grain tools are available, the ability to flexibly and efficiently measure traffic at different, and especially fine-grain, resolutions is limited or non-existent.

For instance, we are unable to answer basic questions such as: what is the distribution of traffic bursts? At which time-scale did the traffic exhibit burstiness? With the identification of long-range dependence (LRD) in network traffic [24], the research community has undergone a mental shift from Poisson and memory-less processes to LRD and bursty processes. Despite its widespread use, however, LRD analysis is hindered by our inability to estimate its parameters unambiguously. Thus, our larger goal is to use fine-grain measurement techniques for fine-grain traffic modeling.

While it is not difficult to choose a small number of preset resolutions and perform measurements for those, the more difficult and useful problem is to support traffic measurements for *all time scales*. Not only do measurement resolutions of interest vary with time (as in burst detection), but in many applications they only become critical *after the fact*, that is, after the measurements have already been performed. This chapter describes an end-host bandwidth measurement tool that succinctly summarizes bandwidth

information and yet answers general queries at arbitrary resolutions without maintaining state for all time scales.

Some representative queries (among many) that we wish such a tool to support are the following:

1. What is the *maximum* bandwidth used at time scale t?
2. What is the *standard deviation* and 95th percentile of the bandwidth at time scale t?
3. What is the *coarsest* time scale at which bandwidth exceeds threshold L?

In these queries, the query parameters t or L are chosen *a posteriori* — after all the measurements have been performed, and thus require supporting all possible resolutions and bandwidths.

Existing techniques: All the above queries above can be easily answered by keeping the entire packet trace. However, our data structures take an order of magnitude less storage than a packet trace (even a sampled packet trace) and yet can answer flexible queries with good accuracy. Note that standard summarization techniques (including simple ones like SNMP packet counters [9]) and more complex ones (e.g., heavy-hitter determination [35]) are very efficient in storage but must be targeted towards a particular purpose and at a fixed time scale. Hence, they cannot answer flexible queries for arbitrary time scales.

Note that sampling 1 in N packets, as in Cisco NetFlow [1], does not provide a good solution for bandwidth measurement queries. Consider a 10 Gbps link with an average packet size of 1000 bytes. This link can produce 10 million packets per second. Suppose the scheme does 1 in 1000 packet sampling. It can still produce 10,000 samples per second with say 6 bytes per sample for time-stamp and packet size. To identify bursts of 1000 packets of 1500 bytes each (1.5 MB), any algorithm would look for intervals containing 1 packet and scale up by the down sampling factor of 1000. The major problem is that this causes false positives. If the trace is well-behaved and has no bursts in any specified period (say 10 msec), the scaling scheme will still falsely identify 1 in 1000 packets as being part of bursts because of the large scaling factor needed for data reduction. Packet sampling, fundamentally, takes no account of the passage of time.

From an information-theoretic sense, packet traces, are inefficient representations for bandwidth queries. Viewing a trace as a time series of point masses (bytes in each packet), it is more memory-efficient to represent the trace as a series of *time intervals* with bytes sent per interval. But this introduces the new problem of choosing the intervals for representation so that bandwidth queries on any interval (chosen after the trace has been summarized) can be answered with minimal error.

Our first scheme builds on the simple idea that for any fixed sampling interval, say 100 microseconds, one can easily compute traffic statistics such as $\max$ or *Standard Deviation* by a few counters each. By exponentially increasing the sampling interval, we can span an aggregation period of length T, and still compute statistics at all time scales from microseconds to milliseconds, using only $O(\log T)$ counters. We call this approach Exponential Bucketing (`EXPB`). The challenge in `EXPB` is to avoid updating all $\log T$ counters on each packet arrival and to prove error bounds.

Our second idea, dubbed Dynamic Bucket Merge (`DBM`), constructs an approximate streaming histogram of the traffic so that bursts stand out as peaks in this histogram. Specifically, we adaptively partition the traffic into k intervals/buckets, in such a way that the periods of heavy traffic map to more refined buckets than those of low traffic. The time-scales of these buckets provide a "visual history" of the burstiness of the traffic—the narrower the bucket in time, the burstier the traffic. In particular, `DBM` is well-suited for identifying not only whether a burst occurred, but *how many bursts*, and *when*.

System Deployment: Exponential Bucketing and Dynamic Bucket Merge have low computational and storage overheads, and can be implemented at multi-gigabit speeds in software or hardware. As shown in Figure 3.1, we envision a deployment scenario where both end hosts and network devices record fine-grain bandwidth summaries to a centralized log server. We argue that even archiving to a single commodity hard disk, administrators could pinpoint, to the second, the time at which correlated bursts occurred on given links, even up to a year after the fact.

This data can be indexed using a relational database, allowing administrators to query bandwidth statistics across links and time. For example, administrators could issue queries to "Find all bursts that occurred between 10 and 11 AM on all links in

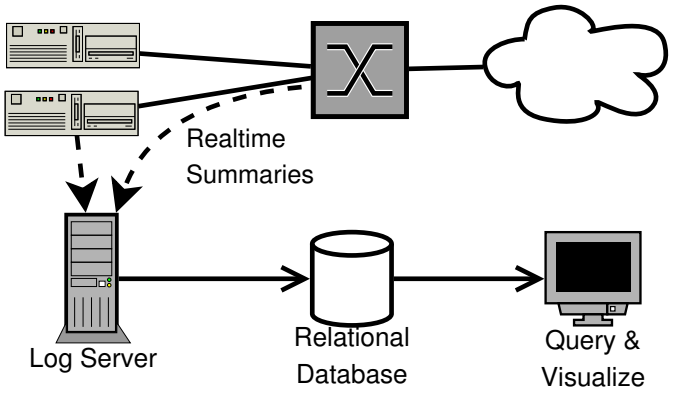


Figure 3.1: Example Deployment. End hosts and network devices implementing `EXPB` and `DBM` push output data over the network to a log server. Data at the server can be monitored and visualized by administrators then collapsed and archived to long-term, persistent storage.

Set S". Set S could be the set of input links to a single switch (which can reveal Incast problems) or the path between two machines. Bandwidth for particular links can then be visualized to further delineate burst behavior. The foundation for answering such queries is the ability to efficiently and succinctly summarize the bandwidth usage of a trace in real-time, the topic of this chapter.

We break down the remainder of our work as follows. We begin with a discussion of related algorithms and systems in Section 3.1. Section 3.2 illustrates the Dynamic Bucket Merge and Exponential Bucketing algorithms, both formally and with examples. We follow with our evaluations in Section 3.3, describe the implications for a system like Figure 3.1 in Section 3.4, and conclude in Section 3.5.

3.1 Related Work

Tcpdump [10] is a mature tool that captures a full log of packets at the end host, which can be used for a wide variety of statistics, including bandwidth at any time scale. While flexible, tcpdump consumes too much memory for continuous monitoring at high speeds across every link and for periods of days. Netflow [1] can capture packet headers in routers but has the same issues. While sampled Netflow reduces storage,

configurations with substantial memory savings cannot detect bursts without resulting in serious false positives. SNMP counters [9], on the other hand, provide packet and byte counts but can only return values at coarse and fixed time scales.

There are a wide variety of summarization data structures for traffic streams, many of which are surveyed in [40]. None of these can directly be adapted to solve the bandwidth problem at all time scales, though solutions to quantile detection do solve some aspects of the problem [40]. For example, classical heavy-hitters [35] measures the heaviest traffic flows during an interval. By contrast, we wish to measure "heavy-hitting sub-intervals across time", so to speak. However, heavy-hitter solutions are complementary in order to identify flows that cause the problem. The LDA data structure [34] is for a related problem – that of measuring average *latency*. LDA is useful for directly measuring latency violations. Our algorithms are complementary in that they help analyze the bandwidth patterns that *cause* latency violations.

`DBM` is inspired by the adaptive space partitioning scheme of [30], but is greatly simplified, and also considerably more efficient, due to the time-series nature of packet arrivals.

3.2 Algorithms

Suppose we wish to perform bandwidth measurements during a time window $[0, T]$, assuming, without loss of generality, that the window begins at time zero. We assume that during this period N packets are sent, with p_i being the byte size of the ith packet and t_i being the time at which this packet is logged by our monitoring system, for $i = 1, 2, \ldots, N$. These packets are received and processed by our system as a *stream*, meaning that the ith packet arrives before the jth packet, for any $i < j$.

The *bandwidth* is a rate, and so converting our observed sequence of N packets into a quantifiable bandwidth usage requires a *time scale*. Since we wish to measure bandwidth at different time scales, let us first make precise what we mean by this. Given a time scale (or *granularity*) Δ, where $0 < \Delta < T$, we divide the measurement window $[0, T]$ into sub-intervals of length Δ, and aggregate all those packets that are sent within the same interval. In this way, we arrive at a sequence $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i

is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals.[1]

Therefore, every choice of Δ leads to a corresponding sequence S_Δ, which we interpret as the bandwidth use at the temporal granularity Δ. All statistical measurements of bandwidth usage at time scale Δ correspond to statistics over this sequence S_Δ. For instance, we can quantify the statistical behavior of the bandwidth at time scale Δ by measuring the *mean, standard deviation, maximum, median, quantiles*, etc. of S_Δ.

In the following, we describe two schemes that can estimate these statistics for *every a posteriori* choice of the time scale Δ. That is, after our algorithms have processed the packet stream, the users can *query* for an arbitrary granularity Δ and receive provable quality approximations of the statistics for the sequence S_Δ.

Our first scheme, `DBM`, is time scale agnostic, and essentially maintains a streaming histogram of the values $s_1, s_2, \ldots, s_k$, by adaptively partitioning the period $[0, T]$. Our second scheme `EXPB` explicitly computes statistics for *a priori* settings of Δ, and then uses them to approximate the statistics for the queried value of Δ.

Since the two schemes are quite orthogonal to each other, it is also possible to use them both in conjunction. We give worst-case error guarantees for both of the schemes. Both schemes are able to compute the mean with perfect accuracy and estimate the other statistics, such as the maximum or standard deviation, with a bounded error. The approximation error for the `DBM` scheme is expressed as an additive error, while the `EXPB` scheme offers a multiplicative relative error. In particular, for the `DBM` scheme, the estimation of the maximum or standard deviation is bounded by an error term of the form $O(\varepsilon B)$, where $0 < \varepsilon < 1$ is a user-specified parameter dependent on the memory used by the data structure, and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window. In the following, we describe and analyze the `DBM` scheme, followed by a description and analysis of the `EXPB` scheme.

3.2.1 Dynamic Bucket Merge

`DBM` maintains a partition of the measurement window $[0, T]$ into what we call

[1] To deal with the boundary problem properly, we assume that each sub-interval includes its right boundary, but not the left boundary. If we assume assume that no packet arrives at time 0, we can form a proper non-overlapping partition this way.

buckets. In particular, an m-bucket partition $\{b_1, b_2, \ldots, b_m\}$, is specified by a sequence of time instants $t(b_i)$, with $0 < t(b_i) \leq T$, with the interpretation that the bucket b_i spans the interval $(t(b_{i-1}),\ t(b_i)]$. That is, $t(b_i)$ marks the time when the ith bucket ends, with the convention that $t(b_0) = 0$, and $t(b_m) = T$. The number of buckets m is controlled by the memory available to the algorithm and, as we will show, the approximation quality of the algorithm improves linearly with m. In the following, our description and analysis of the scheme is expressed in terms of m. Each bucket maintains $O(1)$ information, typically the statistics we are interested in maintaining, such as the total number of bytes sent during the bucket. In particular, in the following, we use the notation $p(b)$ to denote the total number of data bytes sent during the interval spanned by a bucket b.

The algorithm processes the packet stream p_1, p_2, ..., p_N in arrival time order, always maintaining a partition of $[0, T]$ into at most m buckets. (In fact, after the first m packets have been processed, the number of buckets will be exactly m, and the most recently processed packet lies in the last bucket, namely, b_m.) The basic algorithm is quite straightforward. When the next packet p_j is processed, we place it into a new bucket b_{m+1}, with time interval (t_{j-1}, T)—recall that t_{j-1} is the time stamp associated with the preceding packet p_{j-1}. We also note that the right boundary of the predecessor bucket b_m now becomes t_{j-1} due to the addition of the bucket b_{m+1}. Since we now have $m + 1$ buckets, we merge two *adjacent* buckets to reduce the bucket count down to m. Several different criteria can be used for deciding which buckets to merge, and we consider some alternatives later, but in our basic scheme we merge the buckets based on their *packet mass*. That is, we merge two adjacent buckets whose sum of the packet mass is the smallest over all such adjacent pairs. A pseudo-code description of `DBM` is presented in Algorithm 6.

DBM Example

To clarify the operation of `DBM` we give the following example, illustrated in Figure 3.2.

Suppose that we run `DBM` with 4 buckets ($m = 4$), each of which stores a *count* of the number of buckets that have been merged into it, the *sum* of all bytes belonging to it, and the *max* number of bytes of any bucket merged into it. Now suppose that 4

Algorithm 6: DBM

```
1 foreach p_j ∈ S do
2     Allocate a new bucket b_i and set p(b_i) = p_j
3     if i == m + 1 then
5         Merge the two adjacent b_w, b_{w+1} for which p(b_w) + p(b_{w+1}) is
          minimum;
6     end
7 end
```

packets have arrived with masses 10, 20, 35, and 5, respectively. The state of DBM at this point is shown at the top of Figure 3.2. Note that Algorithm 6 required that we merge the buckets with the minimum combined sum. Hence, we maintain a min heap that stores the sums of adjacent buckets.

When a fifth packet with a mass of 40 arrives, DBM allocates a new bucket for it and updates the heap with the sum of the new bucket and its neighbor.

In the final step, the minimum sum is pulled from the heap and the buckets contributing to that sum are merged. In this example, the bucket containing mass 10 and 20 are merged into a single bucket with a new mass of 30 and a max bucket value of 20. Note that we also update the values in the heap, which included the mass of either of the merge buckets.

DBM Analysis

The key property of DBM is that it can estimate the total number of bytes sent during *any* time interval. In particular, let $[t, t']$ be an arbitrary interval, where $0 \leq t, t' \leq T$, and let $p(t, t')$ be the total number of bytes sent during it, meaning $p(t, t') = \sum_{i=1}^{N} \{p_i \mid t \leq t_i \leq t'\}$. Then we have the following result.

Lemma 2. *The data structure* DBM *estimates* $p(t, t')$ *within an additive error* $O(B/m)$*, for any interval* $[t, t']$*, where* m *is the number of buckets used by* DBM *and* $B = \sum_{i=1}^{N} p_i$ *is the total packet mass over the measurement window* $[0, T]$.

Proof. We first note that in DBM each bucket's packet mass is at most $2B/(m-1)$, unless

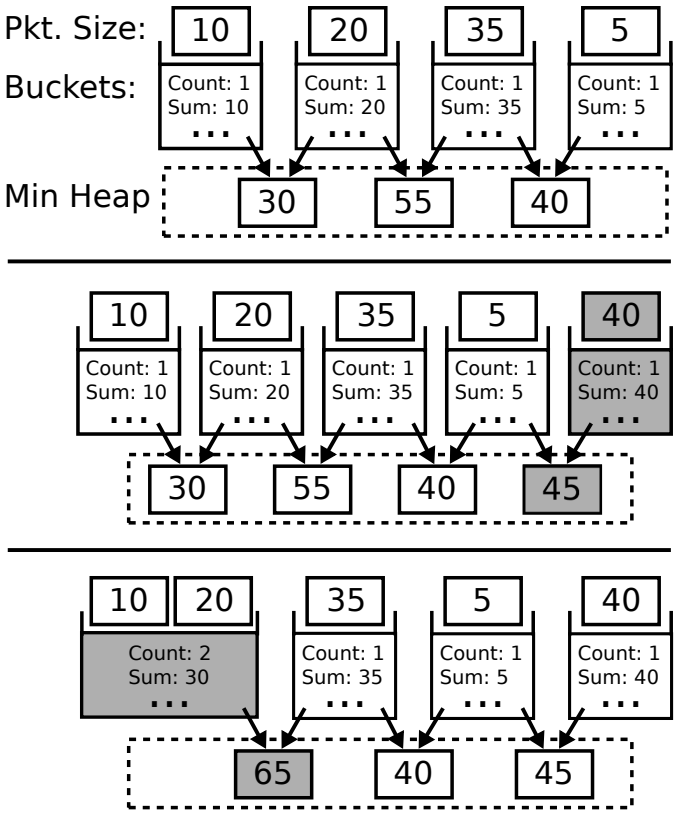


Figure 3.2: Dynamic Bucket Merge with 4 buckets. Initially each bucket contains a single packet and the min heap holds the sums of adjacent bucket pairs. When a new packet (value = 40) arrives, a 5th bucket is allocated and a new entry added to the heap. In the merge step, the smallest value (30) is popped from the heap and the two associated buckets are merged. Last, we update the heap values that depended on either of the merged buckets.

the bucket contains a single packet whose mass is strictly larger than $2B/(m-1)$. In particular, we argue that whenever two buckets need to be merged, there always exists an adjacent pair with total packet mass less than $2B/(m-1)$. Suppose not. Then, summing the sizes of all $(m-1)$ pairs of adjacent buckets must produce a total mass strictly larger than $2(m-1)B/(m-1) = 2B$, which is impossible since in this sum each bucket is counted at most twice, so the total mass must be less than $2B$.

With this fact established, the rest of the lemma follows easily. In order to estimate $p(t, t')$, we simply add up the buckets whose time spans intersect the interval

$[t, t']$. Any bucket whose interval lies entirely inside $[t, t']$ is accurately counted, and so the only error of estimation comes from the two buckets whose intervals only partially intersect $[t, t']$—these are the buckets containing the endpoints t and t'. If these buckets have mass less than $2B/(m-1)$ each, then the total error in estimation is less than $4B/m$, which is $O(\frac{B}{m})$. If, on the other hand, either of the end buckets contains a single packet with large mass, then that packet is correctly included or excluded from the estimation, depending on its time stamp, and so there is no estimation error. This completes the proof. □

Theorem 3. *With* DBM *we can estimate the maximum or the standard deviation of* S_Δ *within an additive error* εB*, using memory* $O(1/\varepsilon)$.

Proof. The proof for the maximum follows easily from the preceding lemma. We simply query DBM for time windows of length Δ, namely, $(i\Delta, (i+1)\Delta]$, for $i = 0, 1, \ldots, \lceil T/\Delta \rceil$, and output the maximum packet mass estimated in any of those intervals. In order to achieve the target error bound, we use $m = \frac{4}{\varepsilon} + 1$ buckets.

We now analyze the approximation of the standard deviation. Recall that the sequence under consideration is $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, for some time scale Δ, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals. Let $Var(S_\Delta)$, $E(S_\Delta)$, and $E(S_\Delta^2)$, respectively, denote the variance, mean, and mean of the squares for S_Δ. Then, by definition, we have

$$Var(S_\Delta) \;=\; E(S_\Delta^2) - E(S_\Delta)^2 = \frac{\sum_{i=1}^{k} {s_i}^2}{k} - E(S_\Delta)^2$$

Since DBM estimates each s_i within an additive error of εB, our estimated variance respects the following bound:

$$\leq \frac{\sum (s_i + \varepsilon B)^2}{k} - E(S_\Delta)^2$$

However, we can compute $E(S_\Delta)^2$ exactly, because it is just the square of the mean. In order to derive a bound on the error of the variance, we assume that $k > m$, that is, the size of the sequence S_Δ is at least as large as the number of buckets in DBM. (Naturally, statistical measurements are meaningless when the sample size becomes too small.) With this assumption, we have $2/k < 2/m$, and since $\varepsilon = 4/(m-1)$, we get

that $2\frac{\sum s_i}{k} \leq \varepsilon B$, which, considering $k \geq 1$, yields the following upper bound for the estimated variance:

$$\leq \frac{\sum s_i^2}{k} - E(S_\Delta)^2 + \frac{k+1}{k}\varepsilon^2 B^2 \;\leq\; Var(S_\Delta) + 2\varepsilon^2 B^2$$

which implies the claim. □

Similarly, we can show the following result for approximating quantiles of the sequence S_Δ.

Theorem 4. *With* DBM *we can estimate any quantile of S_Δ within an additive error εB, using memory $O(1/\varepsilon)$.*

Proof. Let $s_1, s_2, \ldots, s_k$ be the sequence of data in the intervals $(i\Delta, (i+1)\Delta]$, for $i = 1, 2, \ldots, k = \lceil T/\Delta \rceil$, sorted in increasing order, and let $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ be the sorted estimated sequence for the same intervals. We now compute the desired quantile, for instance the 95th percentile, in this sequence. Supposing the index of the quantile is q, we return $\hat{s}_q$. We argue that the error of this approximation is $O(\varepsilon B)$. We do this by estimating bounds on the s_i values that are erroneously (due to approximation) misclassified, meaning reported below or equal the quantile when they are actually larger or vice versa. If no s_i have been misclassified then $\hat{s}_q$ and s_q correspond to the same sample, and by Lemma 2 the estimated value $\hat{s}_q - s_q \leq \varepsilon B$, hence the claim follows. On the other hand, if a misclassification occurred, then the sample s_q is reported at an index different than q in the estimated sequence. Assume without loss of generality that the sample s_q has been reported as $\hat{s}_u$ where $u > q$. Then, by the pigeonhole principle, there is at least a sample s_h ($h > q$) that is reported as $\hat{s}_d$, $d \leq q$. By Lemma 2, $\hat{s}_d - s_h \leq \varepsilon B$. Since s_q and s_h switched ranks in the estimated sequence $\hat{s}$, by Lemma 2 it holds that $s_h - s_q \leq \varepsilon B$ and $\hat{s}_u - \hat{s}_d \leq \varepsilon B$. By assumption $u > q \geq d$, then it follows that $\hat{s}_u \geq \hat{s}_q \geq \hat{s}_d$ in the sorted sequence $\hat{s}$, which implies that $\hat{s}_q - \hat{s}_d \leq \varepsilon B$. The chain of inequalities implies that $\hat{s}_q - s_q \leq 3\varepsilon B$, which completes the proof. □

Algorithm 6 can be implemented at the worst-case cost of $O(\log m)$ per packet, with the heap operation being the dominant step. The memory usage of DBM is $\Theta(m)$ as each bucket maintains $O(1)$ information.

Extensions to DBM for better burst detection

Generic DBM is a useful oracle for estimating bandwidth in any interval (chosen after the fact) with bounded additive error. However, one can tune the merge rule of DBM if the goal is to pick out the bursts only. Intuitively, if we have an aggregation period with k bursts for small k (say 10) spread out in a large interval, then ideally we would like to compress the large trace to k high-density intervals. Of course, we would like to also represent the comparatively low traffic adjacent intervals as well, so an ideal algorithm would partition the trace into $2k + 1$ intervals where the bursts and ideal periods are clearly and even visually identified. We refer to the generic scheme discussed earlier that uses *merge-by-mass* as DBM−mm, and describe two new variants as follows.

- merge-by-variance (DBM−mv): merges the two adjacent buckets that have the minimum aggregated packet mass variance
- merge-by-range (DBM−mr): merges the two adjacent buckets that have the minimum aggregated packet mass range (defined as the difference between maximum and minimum packet masses within the bucket)

These merge variants can also be implemented in logarithmic time, and require storing $O(1)$ additional information for each bucket (in addition to $p(b_i)$).

One minor detail is that DBM−mv and DBM−mr are sensitive to null packet mass in an interval while DBM−mm is not. For these reasons, we make the DBM−mr and DBM−mv algorithms work on the sequence defined by S_Δ, where Δ is the minimum time scale at which bandwidth measurements can be queried. Then DBM−mr and DBM−mv represents S_Δ as a histogram on m buckets, where each bucket has a discrete value for the signal. The goal of a good approximation is to minimize its predicted value versus the true under some error metric. We consider both the L_2 norm and the L_∞ norm for the approximation error.

$$E_2 = (\sum_{i=1}^{n} |s_i - \hat{s}_i|^2)^{\frac{1}{2}} \tag{3.1}$$

where $\hat{s}_i$ is the approximation for value s_i.

$$E_\infty = \max_{i=1}^{n} |s_i - \hat{s}_i| \tag{3.2}$$

We compare the performance of `DBM-mr` and `DBM-mv` algorithms with the optimal offline algorithms, that is, a bucketing scheme that would find the optimal partition of S_Δ to minimize the E_2 or the E_∞ metric. Then, the analysis of [17, 27] can be adapted to yield the following results that formally state our intuitive goal of picking out m bursts with $2m + 1$ pieces of memory.

Theorem 5. *The L_∞ approximation error of the m-bucket* `DBM-mr` *is never worse than the corresponding error of an optimal $m/2$-bucket partition.*

Theorem 6. *The L_2 approximation error of the m-bucket* `DBM-mv` *is at most $\sqrt{2}$ times the corresponding error of an optimal $m/4$-bucket partition.*

3.2.2 Exponential Bucketing

Our second scheme, which we call Exponential Bucketing (`EXPB`), explicitly computes statistics for *a priori* settings of $\Delta_1, \ldots, \Delta_m$, and then uses them to approximate the statistics for the queried value for *any* Δ, for $\Delta_1 \leq \Delta \leq \Delta_m$. We assume that the time scales grow in powers of two, meaning that $\Delta_i = 2^{i-1}\Delta_1$. Therefore, we can assume that the scheme processes data at the time scale Δ_1, namely, the sequence $S_{\Delta_1} = (s_1, s_2, \ldots, s_k)$.

Conceptually, `EXPB` maintains statistics for all m time scales $\Delta_1, \ldots, \Delta_m$. A naïve implementation would require updating $O(m)$ counters per (aggregated) packet s_i. However, by carefully orchestrating the accumulator update when a new s_i is available it is possible to avoid spending m updates per measurement as shown in Algorithm 7.

The intuition is as follows. Suppose one is maintaining statistics at 100 μs and 200 μs intervals. When a packet arrives, we update the 100 μs counter but not the 200 μs counter. Instead, the 200 μs counter is updated only when the 100 μs counter is zeroed. In other words, only the lowest granularity counter is updated on every packet, and coarser granularity counters are only updated when all the finer granularity counters are zeroed.

Algorithm 7: EXPB

```
1  sum=< 0, ..., 0 > (m times) ;
2  foreach s_i do
3      sum[0]=s_i;
4      j=0;
6      repeat
8          updatestat(j,sum[j]);
9          if j < m then
10             sum[j+1]+=sum[j];
11         end
12         sum[j]=0;
13         j++;
14     until i mod 2^j ≠ 0 or j ≥ m ;
15 end
```

EXPB Example

To better understand the EXPB algorithm we now present the example illustrated in Figure 3.3.

In this example, we maintain 3 buckets ($m = 3$) each of which stores statistics at time scales of 1, 2 and 4 time units. Each bucket stores the *count* of the intervals elapsed, the *sum* of the bytes seen in the current interval, and fields to compute max and standard deviation. We label the time units along the top and the number of bytes accumulated during each interval along the bottom.

In the first time interval 10 bytes are recorded in the first bucket and 10 is pushed to the sum of the second bucket. We repeat this operation when 20 is recorded in the second interval. Since 2 time units have elapsed, we also update the statistics for the Δ_2 time scale, and add bucket two's sum to bucket 3. In the third interval we update bucket 1 as before. Finally, at time 4 we update bucket 2 with the current sum from bucket 1, update bucket two's statistics, and push bucket two's sum to bucket 3. Finally, we update the statistics for Δ_3 with bucket three's sum.

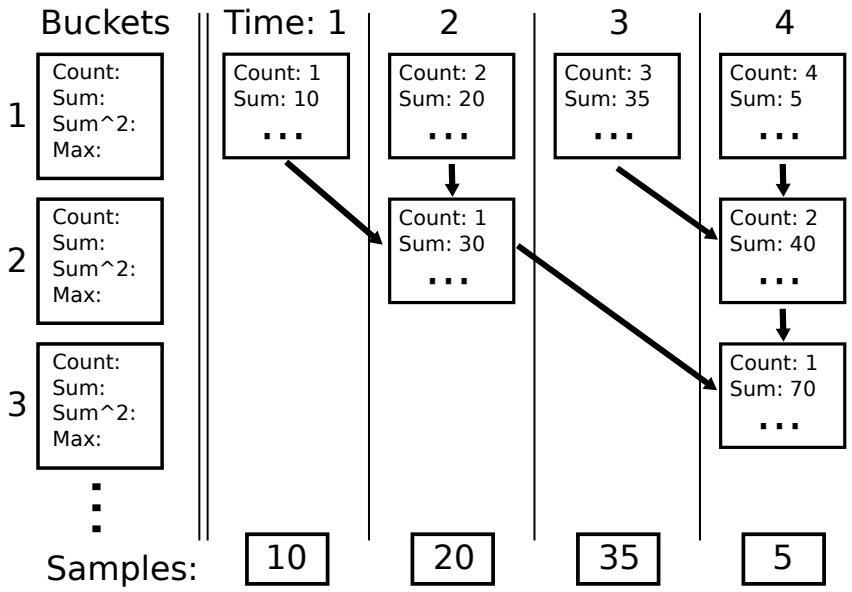


Figure 3.3: Exponential Bucketing Example. Each of the m buckets collects statistics at 2^{i-1} times the finest time scale. At the end of each time scale, Δ_i, buckets 1 to i must be updated. Before storing the new sum in a bucket j, we first add the old sum into bucket $j+1$, if it exists.

EXPB Analysis

Algorithm 7 uses $O(m)$ memory and runs in $O(k)$ worst-case time, where $k = \lceil T/\Delta_1 \rceil$ is the number of intervals at the lowest time scale of the algorithm. The per-interval processing time is amortized constant, since the repeat loop starting at Line 6 simply counts the number of trailing zeros in the binary representation of i, for all $0 < i < k = T/\Delta$. The procedure $updatestat()$ called at Line 8 updates in constant time the $O(1)$ information necessary to maintain the statistics for each Δ_i, for $1 \leq i \leq m$.

We now describe and analyze the bandwidth estimation using `EXPB`. Given any query time scale Δ, we output the maximum of the bandwidth corresponding to the smallest index j for which $\Delta_j \geq \Delta$, and use the sum of squared packet masses stored for granularity Δ_j to compute the standard deviation. The following lemma bounds the error of such an approximation.

Lemma 3. *With* `EXPB` *we can return an estimation of the maximum or standard deviation of S_Δ that is between factor $1/2$ and 3 from the true value. The bound on the*

standard deviation holds in the limit when the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ is large.

Proof. We first prove the result for the statistic maximum, and then address the standard deviation. Let I be the interval $((i-1)\Delta, i\Delta]$ corresponding to the time scale Δ in which the maximum value is achieved, and let $p(I)$ be this value. Since $\Delta_j \geq \Delta$, there are at two most consecutive intervals I_i^j, I_{i+1}^j at time scale Δ_j that together cover I. By the pigeonhole principle, either I_i^j or I_{i+1}^j must contain at least half the mass of I, and therefore the maximum value at time scale Δ_j is at least 1/2 of the maximum value at Δ. This proves the lower bound side of the approximation. In order to obtain a corresponding upper bound, we simply observe that if I_i^j is the interval at time scale Δ_j with the maximum value, then I_i^j overlaps with at most 3 intervals of time scale Δ. Thus, the maximum value at time scale Δ_j cannot be more than 3 times the maximum at Δ proving an upper bound on the approximation.

The analysis for the standard deviation follows along the same lines, using the observation that $stddev_\Delta = \sqrt{E(S_\Delta^2) - E(S_\Delta)^2}$. An argument similar to the one used for the maximum value holds for the approximation of $E(S_\Delta^2)$. Then assuming the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ to be a constant sufficiently greater than 1 implies the claim. We omit the simple algebra from this extended abstract. □

We note that there is a non-trivial extension of `EXPB` which allows it to work with a set of exponentially increasing time granularities whose common ratio can be any $\alpha > 1$. This can reduce average error. For a general $\alpha > 1$, Algorithm 7 cannot be easily adapted, so we need a generalization of it that uses an event queue while processing measurements to schedule when in the future a new measurement of length Δ_j must be sent to $updatestat()$. The details are omitted for lack of space.

3.2.3 Culprit Identification

As mentioned earlier, we do not want to simply identify bursts but also to *identify the flow* (e.g., TCP connection, or source IP address, protocol) that caused the burst so that the network manager can reschedule or move the offending station or application. The naive approach would be to add a heavy-hitters [35] data structure to each DBM

bucket, which seems expensive in storage. Instead, we modify DBM to include two extra variables per bucket: a flowID and a flow count for the flowID.

The simple heuristic we suggest is as follows. Initially, each packet is placed in a bucket, and the bucket's flowID is set to the flowID of its packet. When merging two buckets, if the buckets have the same flowID, then that flowID becomes the flowID of the merged bucket and the flow counts are summed. If not, then one of the two flowIDs is picked with probability proportional to their flow counts. Intuitively, the higher count flows are more likely to be picked as the main contributor in each bucket as they are more likely to survive merges.

For EXPB, a simple idea is to use a standard heavy-hitters structure [35] corresponding to each of the logarithmic time scales. When each counter is reset, we update the flowID if the maximum value has changed and reinitialize the heavy-hitters structure for the next interval. This requires only a logarithmic number of heavy-hitters structures. Since there appears to be redundancy across the structures at each time scale, more compression appears feasible but we leave this for future work.

3.3 Evaluation

We now evaluate the performance and accuracy of DBM and EXPB to show that they fulfill our goal of a tool that efficiently utilizes memory and processing resources to faithfully capture and display key bandwidth measures. We will show that DBM and EXPB use significantly fewer resources than packet tracing and are suitable for network-wide measurement and visualization.

3.3.1 Measurement Accuracy

We implemented EXPB and the three variants of DBM as user-space programs and evaluated them with real traffic traces. Our traces consisted of a packets captured from the 1 Gigabit switch that connects several infrastructure servers used by the Systems and Networking group at U.C. San Diego, and socket-level send data produced by the record-breaking TritonSort sorting cluster [43].

Our "rsync" trace captured individual packets from an 11-hour period during

which our NFS server ran its monthly backup to a remote machine using rsync. This trace recorded the transfer of 76.2 GB of data in 60.6 million packets, of which 66.6 GB was due to the backup operation. The average throughput was 15.4 Mbps with a maximum of 782 Mbps for a single second.

The "tritonsort" trace contains time-stamped byte counts from successful `send` system calls on a single host during the sorting of 500 GB of data using 23 nodes connected by a 10 Gbps network. This trace contains an average of 92,488 send events per second, with a peak of 123,322 events recorded in a single 1-second interval. In total, 20.8 GB were transferred over 34.24 seconds for an average throughput of 4.9 Gbps.

Ideally, our evaluation would include traffic from a mix of production applications running over a 10 Gbps network. While we do not have access to such a deployment, our traces provide insight into how `DBM` and `EXPB` might perform given the high bandwidth and network utilization of the "tritonsort" trace and the large variance in bandwidth from second to second in the "rsync" trace.

For our accuracy evaluation, we used an aggregation period of 2 seconds. To avoid problems with incomplete sampling periods in `EXPB`, we must choose our time scales such that they all evenly divide our aggregation period. Since the prime factors of 2 seconds in nsec are 2^{11} and 5^{10} nsec, `EXPB` can use up to 11 buckets. Thus for `EXPB`, we choose the finest time scale to be $\Delta = 78.125$ μs (5^7 nsec) and the coarsest to be $\Delta = 80$ msec ($2^{11}5^7$ nsec), which is consistent with the time scales for interesting bursts in data centers. For consistency, we also configure `DBM` to use a base sampling interval of 78.125 μs, but note that it can answer queries up to $\Delta = 2$ seconds.

To provide a baseline measurement, we computed bandwidth statistics for all of our traces at various time scales where $\Delta \geq 78.125$ μs. To ensure that all measurements in S_Δ are equal, we only evaluated time scales that evenly divided 2 seconds. In total, this provided us with ground-truth statistics at 52 different time scales ranging from 78.125 μs to 2 seconds. In the following sections we report accuracy in terms of error relative to these ground-truth measurements. While any number of values could be used for Δ and T in practice, we used these values across our experiments for the sake of a consistent and representative evaluation between algorithms.

Accuracy vs. Memory

We begin by investigating the tradeoff between memory and accuracy. At one extreme, SNMP can calculate average bandwidth using only a single counter. In contrast, packet tracing with tcpdump can calculate a wide range of statistics with perfect accuracy, but with storage cost scaling linearly with the number of packets. Both `DBM` and `EXPB` provide a tradeoff between these two extremes by supporting complex queries with bounded error, but with orders of magnitude less memory.

For comparison, consider the simplest event trace, which captures a 64-bit timestamp and a 16-bit byte length for each packet sent or received. Using this data, one could calculate bandwidth statistics for the trace with perfect accuracy at a memory cost of 6 bytes *per event*. In contrast, `DBM` and `EXPB` require 8 and 16 bytes of storage per bucket used, respectively, along with a few bytes of metadata for each aggregation period.

To quantify these differences, we queried our traces for max, standard deviation, and 95th percentile (`DBM` only). For each statistic, we compute the average relative error of the measurements at each of our reference time scales and report the worst-case. To avoid spurious errors due to low sample counts, we omit accuracy data for standard deviations with fewer than 10 samples per aggregation period and 95th percentiles with fewer than 20 samples per aggregation period. We show the tradeoff between storage and accuracy in Table 3.1.

While the simple packet trace gives perfectly accurate statistics, both `DBM` and `EXPB` consume memory at a fixed rate, which can be configured by specifying the number of buckets and the aggregation period. In the presented configuration, both `DBM` and `EXPB` generate 4 KBps and 96 Bps, respectively — orders of magnitude less memory than the simple trace.

The cost of reduced storage overhead in `DBM` and `EXPB` is the error introduced in our measurements. However, we see that the range of average relative error rates is reasonable for max, standard deviation, and 95th percentile measurements. Further, of the `DBM` algorithms, `DBM-mr` gives the lowest errors throughout. While not shown, `DBM`'s errors are largely due to under-estimation, but its accuracy improves as the query interval grows. `EXPB` gives consistent estimation errors for max across all of our reference points, but gradually degrades for standard deviation estimates as query intervals

increase. Thus, for this trace, EXPB achieves the lowest error for query intervals less than 2msec. We have divided Table 3.1 to show the worst-case errors in these regions.

In Table 3.2 , we show the accuracy of DBM−mr and EXPB when run on the "rsync" trace with the same parameters as before. We note that again DBM−mr gives the most accurate results for larger query intervals, but now out-performs EXPB for query intervals greater than 160μs for max and 1msec for standard deviation.

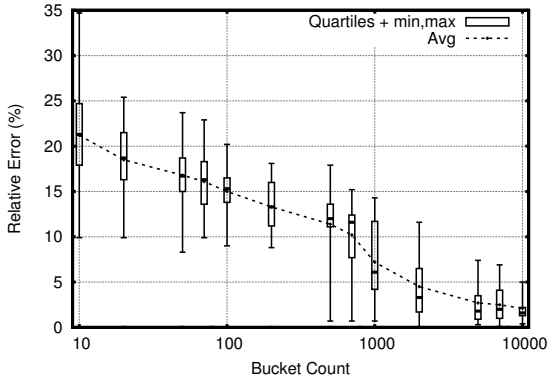
To see the effect of scaling the number of buckets, we picked a representative query interval of 400 μs and investigated the accuracy of DBM−mr as the number of buckets were varied. The results of measuring the max, standard deviation and 95th percentile on the "tritonsort" trace are shown in Figure 3.4. We see that the relative error for all measurements decreases as the number of buckets is increased. However, at 4,000 buckets the curves flatten significantly and additional buckets beyond this do not produce any significant improvement in accuracy. While one might expect the error to drop to zero when the number of buckets is equal to the number of samples at S_Δ (5000 samples for 400μs), we do not see this since the trace is sampled at a finer granularity (78.125 μs) and the buckets are merged online. There is no guarantee that DBM will merge the buckets such that each spans exactly 400μs of the trace.

With approximations of the max and standard deviation with this degree of accuracy, we see both DBM and EXPB as an excellent, low-overhead alternative to packet tracing.

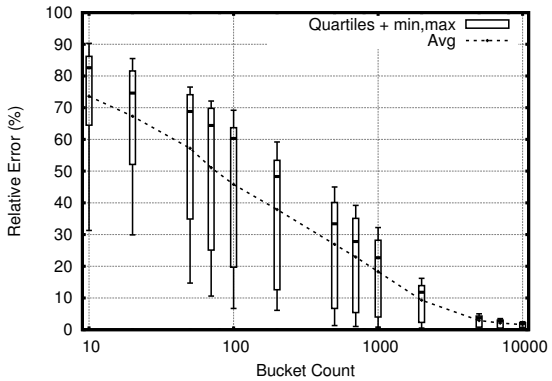
DBM Visualization

One unique property of the DBM algorithms is that they can be visualized to show users the shape of the bandwidth curves. Note that we proved earlier that DBM−mr is optimal in some sense in picking out bursts. We now investigate experimentally how all DBM variants do in burst detection.

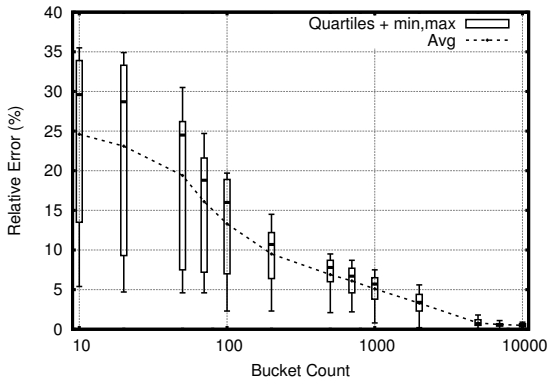
In Figures 3.5 we show the output for a single, 2 second aggregation period from the "rsync" trace using DBM−mr. For visual clarity, we configured DBM−mr to aggregate measurements at a 4 msec base time scale (250 data points) using 9 buckets. Figure 3.5 shows the raw data points (bandwidth use in each 4 msec interval of the 2 second trace) with the DBM−mr output superimposed. Notice that DBM−mr picks



(a) Max measurements



(b) Std. Dev. measurements



(c) 95th Percentile measurements

Figure 3.4: Relative error for `DBM-mr` algorithm shown for the 400 μs time scale. The box plots show the median and the range of relative errors from the 25th to 75th percentiles, with the box whiskers indicating the min and max errors.

out four bursts (the vertical lines). The fourth burst looks smaller than the 3.1 Mbps burst observable in the raw trace. This is because there were two adjacent measurement intervals in the raw trace with bandwidths of 3.1 and 2.2 Mbps, respectively. `DBM-mr` merged these measurements into a single bucket of with an average bandwidth of 2.65 Mbps for 8 msec.

We show the output for all `DBM` algorithms in a more clean visual form in Figures 3.6a, 3.6c and 3.6e. We have normalized the width of the buckets and list their start and end times on the x-axis. Additionally, we label each bucket with its mass (byte count). This representation compresses periods of low traffic and highlights short-lived, high-bandwidth events. From the visualization of `DBM-mr` in Figure 3.6e, we can quickly see that there were four periods of time, each lasting between 4 and 8 msec where the bandwidth exceeded 2.3 Mbps. Note that in Figure 3.6a, `DBM-mm` picks out only two bursts. The remaining bursts have been merged into the three buckets spanning the period from 1440 to 1636 msec, thereby reducing the bandwidth (the y-axis) because the total time of the combined bucket increases.

In practice, a network administrator might want to quickly scan such a visualization and look for microburst events. To simulate such a scenario, we randomly inserted three bursts, each lasting 4 msec and transmitting between 4.0 and 4.4 MB of data. We show the `DBM` visualization for this augmented trace in bottom of Figure 3.6. `DBM-mr` and `DBM-mm` both allocate their memory resources to capture all three of these important events, even though they only represent 12 msec of a 2 second aggregation period. Again, `DBM-mr` cleanly picks out the three bursts.

Accuracy at High Load

As mentioned in Lemma 2, the error associated with the `DBM` algorithms increases with the ratio of total packet mass (total bytes) to number of buckets within an aggregation period. We now investigate to what extent increasing the mass within an aggregation period affects the measurement accuracy of `DBM`. To evaluate this, we first configured `DBM` to use a base time scale of $\Delta = 78.125$ μs and 1000 buckets, as before, but vary the mass stored in `DBM` by changing the aggregation period. Figures 3.7a & 3.7b show the change in average relative error for both max and standard deviation

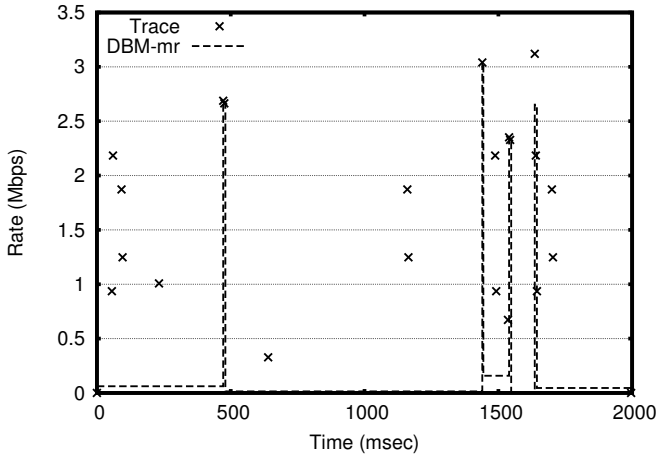


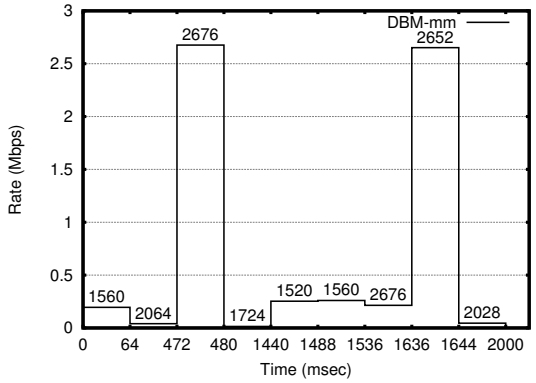
Figure 3.5: Visualization of events from a 2 second aggregation period overlaid with the output of `DBM-mr` using 9 buckets and a 4 msec measurement time scale.

statistics in our high-bandwidth "tritonsort" trace at a representative query time scale (400 μs) as the aggregation period is varied between 1 and 16 seconds.

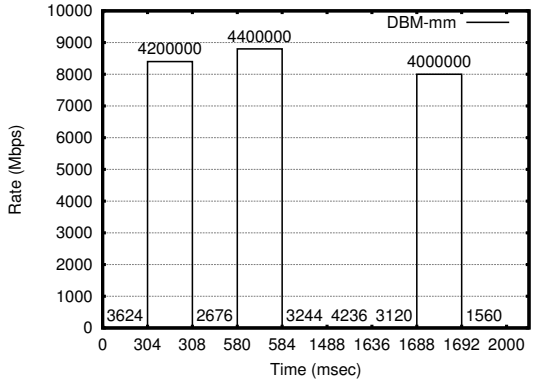
For `DBM-mm` and `DBM-mv` with 1000 buckets the relative error diverges significantly as the aggregation period is increased. In contrast, `DBM-mr` shows only a subtle degradation for max from 5.9% to 12.3%. For standard deviation, `DBM-mv` show consistently poor performance with average relative errors increasing from 32% to 64%, while both `DBM-mm` and `DBM-mr` trend together with `DBM-mr`'s errors ranging from 9.8 to 31.7%.

We contrast `DBM-mr`'s performance for these experiments with that of `EXPB`. We see that `EXPB`'s average relative error in the max measurement gradually falls from 2.8% to 1.9% as the aggregation period increases. Further, the error in standard deviation falls from 1.4% at a 1 second aggregation period to 0.5% at 16 seconds.

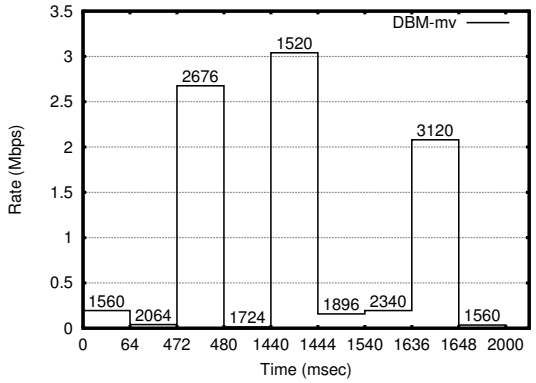
These results indicate that degradation in accuracy does occur as the ratio of the total packet mass to bucket count increases, as predicted by Lemma 2. While `DBM` must be configured correctly to bound the ratio of packet mass to bucket count, `EXPB`'s accuracy is largely unaffected by the packet mass or aggregation period.



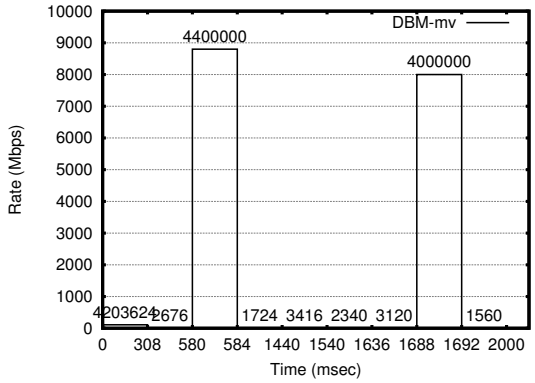
(a) DBM−mm visualization



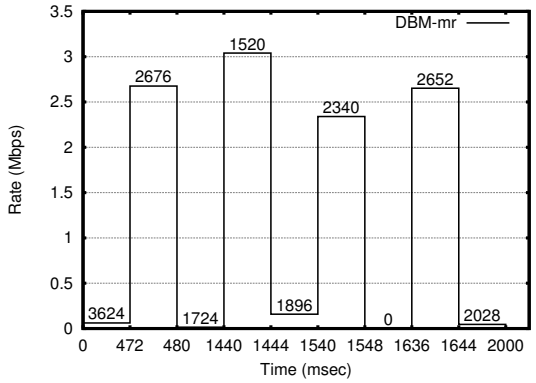
(b) DBM−mm visualization of bursty traffic



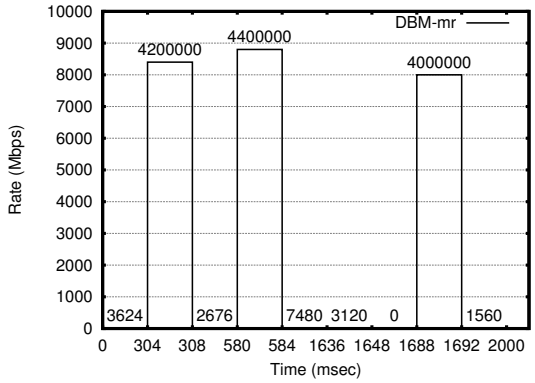
(c) DBM−mv visualization



(d) DBM−mv visualization of bursty traffic

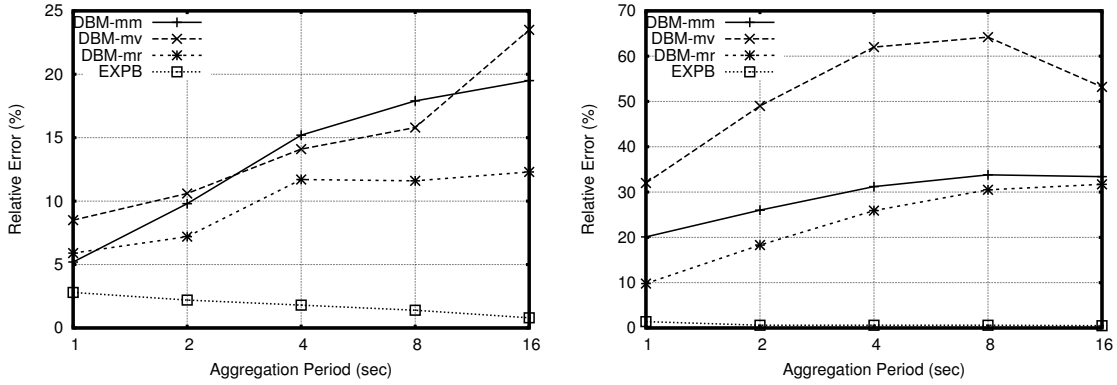


(e) DBM−mr visualization



(f) DBM−mr visualization of bursty traffic

Figure 3.6: Visualization of DBM with 9 buckets over a single 2 second aggregation period. The start and end times for each bucket are shown on the x-axis, and each bucket is labeled with its mass (byte count). The top figures show the various DBM approximations of a single aggregation period, while the lower graphs show the same period with three short-lived, high bandwidth bursts randomly inserted.



(a) Max measurements with 1000 buckets

(b) Std. Dev. measurements with 1000 buckets

Figure 3.7: Average relative error for the `DBM` with 1000 buckets and `EXPB` with 11 buckets shown on the "tritonsort" trace for a 400 μs query interval and various aggregation periods.

3.3.2 Performance Overhead

As previously stated, we seek to provide an efficient alternative to packet capture tools. Hence we compare the performance overhead of `DBM` and `EXPB` to that of an unmodified vanilla kernel, and to the well-established tcpdump[10].

We implemented our algorithms in the Linux 2.6.34 kernel along with a userspace program to read the captured statistics and write them to disk. To provide greater computational efficiency we constrained the base time scale and the aggregation period to be powers of 2. The following experiments were run on 2.27 GHz, quad-core Intel Xeon servers with 24 GB of memory. Each server is connected to a top-of-rack switch via 10 Gbps Ethernet and has a round trip latency of approximately 100 μs.

To quantify the impact of our monitoring on performance, we first ran iperf [5] to send TCP traffic between two machines on our 10 Gbps network for 10 seconds. In addition, we instrumented our code to report the time spent in our routines during the test. We first ran the vanilla kernel source, then added different versions of our monitoring to aggregate 64 μs intervals over 1 second periods. We report both the bandwidth achieved by iperf and the average latency added to each packet at the sending server in Table 3.3. For comparison, we also report performance numbers for tcpdump

when run with the default settings and writing the TCP and IP headers (52 bytes) of each packet directly to local disk. As DBM−mr is nearly identical to DBM−mm with respect to implementation, we omit DBM−mr's results.

As discussed in section 3.2.1, we see that the latency overhead per packet increases roughly as the log of the number of buckets. However, iperf's maximum throughput is not degraded by the latency added to each packet. Since the added latency per packet is several orders of magnitude less than the RTT, the overhead of DBM should not affect TCP's ability to quickly grow its congestion window. In contrast to DBM, tcpdump achieves 3.5% less throughput.

To observe the overhead of our monitoring on an application, we transferred a 1GB file using scp. We measured the wall-clock time necessary to complete the transfer by running scp within the Linux's time utility. To quantify the affects of our measurement on the total completion time, we measured the total overhead imposed on packets as they moved up and down the network stack. We report this overhead as a percentage of each experiment's average completion time (monitoring time divided by scp completion time). Each experiment was replicated 60 times and results are reported in Table 3.4. We see that although the cumulative overhead added by DBM grows logarithmically with the number of buckets, the time for scp to complete increases by at most 4.5%.

We see that our implementations of DBM and EXPB have a negligible impact on application performance, even while monitoring traffic at 10 Gbps.

3.3.3 Evaluation Summary

Our experiments indicate DBM−mr consistently provides better burst detection and has reasonable average case and worst case error for various statistics. When measuring at arbitrary time scales, EXPB has comparable or better average and worst-case error than DBM while using less memory. In addition, EXPB is unaffected by high mass in a given aggregation period. On other hand, DBM can approximate time series, which is useful for seeing how burst are distributed in time and for calculating more advanced statistics (i.e. percentiles). We recommend a parallel implementation where EXPB is used for Max and Standard Deviation and DBM−mr is used for all other queries.

3.4 System Implications

So far, we have described DBM and EXPB as part of an end-host monitoring tool that can aggregate and visualize bandwidth data with good accuracy. However, we see these algorithms a part of a larger infrastructure monitoring system.

Long-term Archival and Database Support It is useful for administrators to retrospectively troubleshoot problems that are reported by customers days after the fact. At slightly more than 4 KBps, the data produced by both DBM and EXPB for a week (2.4 GB per link) could easily be stored to a commodity disk. With this data, an administrator can pinpoint traffic abnormalities at microsecond time scales and look for patterns across links. The data can be compacted for larger time scales by reducing granularity for older data. For example, one hour of EXPB data could be collapsed into one set of buckets containing max and standard deviation information at the original resolutions but aggregated across the hour.

With such techniques, fine-grain network statistics for hundreds of links over an entire year could be stored to a single server. The data could be keyed by link and time and stored in a relational database to allow queries across time (is the traffic on a single link becoming more bursty with time?) or across links (did a number of bursts correlate on multiple switch input ports?).

Hardware Implementation Both DBM and EXPB algorithms can be implemented in hardware for use in switches and routers. EXPB has an amortized cost of two bucket updates per measurement interval. Since bucket updates are only needed at the frequency of the measurement time scale, these operations could be put on a work queue and serviced asynchronously from the main packet pipeline. The key complication for implementing DBM in hardware is maintaining a binary heap. However, a 1000 bucket heap can be maintained in hardware using a 2-level radix-32 heap that uses 32-way comparators at 10 Gbps. Higher bucket sizes and speeds will require pipelining the heap. The extra hardware overhead for these algorithms in gates is minimal. Finally, the logging overhead is very small, especially when compared to NetFlow.

3.5 Conclusions

Picking out bursts in a large amount of resource usage data is a fundamental problem and applies to all resources, whether power, cooling, bandwidth, memory, CPU, or even financial markets. However, in the domain of data center networks, the increase of network speeds beyond 1 Gigabit per second and the decrease of in-network buffering has made the problem one of great interest.

Managers today have little information about how microbursts are caused. In some cases they have identified paradigms such as InCast, but managers need better visibility into bandwidth usage and the perpetrators of microbursts. They would also like better understanding of the temporal dynamics of such bursts. For instance, do they happen occasionally or often? Do bursts linger below a tipping point for a long period or do they arise suddenly like tsunamis? Further, correlated bursts across links lead to packet drops. A database of bandwidth information from across an administrative domain would be valuable in identifying such patterns. Of course, this could be done by logging a record for every packet, but this is too expensive to contemplate today.

Our work provides the first step to realizing such a vision for a cheap network-wide bandwidth usage database by showing efficient summarization techniques at links ($\sim$4 KB per second, for example, for running `DBM` and `EXPB` on 10 Gbps links) that can feed a database backend as shown in Figure 3.1. Ideally, this can be supplemented by algorithms that also identify the flows responsible for bursts and techniques to join this information across multiple links to detect offending applications and their timing. Of the two algorithms we introduce, Exponential Bucketing offers accurate measurement of the average, max and standard deviation of bandwidths at arbitrary sampling resolutions with very low memory. In contrast, Dynamic Bucket Merge approximates a time-series of bandwidth measurements that can be visualized or used to compute advanced statistics, such as quantiles.

While we have shown the application of `DBM` and `EXPB` to bandwidth measurements in end hosts, these algorithms could be easily ported to in-network monitoring devices or switches. Further, these algorithms can be generally applied to any time-series data, and will be particularly useful in environments where resource spikes must be detected at fine time scales but logging throughput and archival memory is constrained.

Chapter 3, in part, is a reprint of the material as it appears in the article "Efficiently Measuring Bandwidth at All Time Scales" in the Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, March 2011. Frank Uyeda, Luca Foschini, Subhash Suri, George Varghese.

Table 3.1: Memory vs. Accuracy. We evaluate the "tritonsort" trace with a base time scale of Δ =78.125 μs and a 2 second aggregation period. Data output rate is reported for a simple packet trace compared with the `DBM` and `EXPB` algorithms. For each statistic, we compute the max of the average relative error of measurements for each of our reference time scales.

		Max of Avg. Relative Error					
		$\leq$ 2 msec			$>$ 2msec		
	Output Rate	Max	S.Dev.	95th	Max	S.Dev.	95th
packet trace (avg)	555 KBps						
(peak)	740 KBps	0%	0%	0%	0%	0%	0%
`DBM-mm`, 1000 buckets	4 KBps	25.9%	43.3%	18.3%	2.2%	5.7%	1.1%
`DBM-mv`, 1000 buckets	4 KBps	16.7%	58.9%	26.7%	7.2%	39.0%	10.4%
`DBM-mr`, 1000 buckets	4 KBps	14.0%	35.0%	16.1%	2.0%	4.1%	0.9%
`EXPB`, 11 buckets	96 Bps	2.7%	2.5%	N/A	2.8%	8.1%	N/A

Table 3.2: We repeated our evaluation with the "rsync" trace and report accuracy results for our two best performing algorithms — `DBM-mr` and `EXPB`. We calculated the average relative error for each of our reference time scale and show the worst case.

		Max of Avg. Rel. Error		
	Output	Max	S.Dev.	95th
trace (avg)	9.2 KBps			
(peak)	396 KBps	0%	0%	0%
`DBM-mr`	4 KBps	7.6%	14.7%	14.9%
`EXPB`	96 Bps	14.2%	5.9%	N/A

Table 3.3: Average TCP bandwidth reported by iperf over 60 10-second runs. We also show the average time spent in the kernel-level monitoring functions for each packet sent. `DBM` and `EXPB` were run with a base time scale of $\Delta = 64$ μs and T = 1 second aggregation period.

Version	Buckets	Avg. BW	Overhead/Pkt
vanilla	N/A	9.053 Gbps	0.0 nsec
`DBM-mm`	10	9.057 Gbps	256.5 nsec
	100	9.010 Gbps	335.7 nsec
	1000	9.104 Gbps	237.5 nsec
	10000	8.970 Gbps	560.4 nsec
`DBM-mv`	10	9.043 Gbps	205.7 nsec
	100	8.986 Gbps	327.9 nsec
	1000	9.067 Gbps	432.2 nsec
	10000	9.067 Gbps	457.2 nsec
`EXPB`	14	9.109 Gbps	169.4 nsec
tcpdump	N/A	8.732 Gbps	N/A

Table 3.4: The time needed to transfer a 1GB file over `scp`. We measured the cumulative overhead incurred by our monitoring routines for all send and receive events. We report this overhead as a percentage of each experiment's total running time.

Version	Buckets	Time	Overhead
vanilla	N/A	14.133 sec	N/A
`DBM-mm`	10	14.334 sec	1.3%
	100	14.765 sec	1.9%
	1000	14.483 sec	2.7%
	10000	14.527 sec	2.5%
`DBM-mv`	10	14.320 sec	1.7%
	100	14.344 sec	2.3%
	1000	14.645 sec	2.9%
	10000	14.482 sec	3.1%
`EXPB`	14	14.230 sec	0.4%
tcpdump	N/A	15.253 sec	7.9%

Chapter 4

Evaluating the Efficacy of Software-Based Traffic Pacing

Recent measurement studies [33, 29] have shown that modern data-center traffic consists of both short latency-critical flows (e.g., search queries) together with large bandwidth-intensive flows (e.g., transactions that build the search index). The latency-critical flows have various service-level agreements (SLA's) that govern acceptable latency: in some trading environments, the requirement is for latencies on the order of microseconds. While TCP and other window-based flow control techniques provide good bandwidth utilization for large flows, they do so by opening large windows until losses occur. As a result switch buffers often fill, delaying latency-sensitive packets. At best, this causes latency increases; at the worst, it causes packet drops, which can be followed by large timeout periods. The situation is exacerbated by the fact that many commodity switches in the data center are shallow-buffered: their cost is reduced by shrinking the amount of costly high-speed buffer memory. In these environments, even short-lived "microbursts" of traffic may significantly delay flows [42, 19]. These factors have led to proposals for pacing at 1 and 10 Gbps.

Pacing and rate-based congestion control has gone in and out of fashion in academic research from NETBLT [20] in 1987 to proposals for ATM flow control [32] in 2000. However, pacing at end hosts is becoming mainstream once again. For example, the Quantized Congestion Notification (QCN) [11] protocol for Data Center Ethernets uses congestion feedback from switches that is then acted upon by Network Interface

Cards (NICs) at end hosts. Far from being a fringe movement, QCN has been approved by the IEEE in the 802.1Qau standard [4] and is actively implemented and vigorously marketed by industry leaders, such as Cisco, HP, and Intel. The increasing use of pacing is driven by higher speeds accompanied by small buffer sizes at switches, and the prevalence of financial and cluster applications with stringent demands on latency and loss.

While rate control with QCN has been deployed within the data center, it requires hardware support at the end host, which may not exist in deployed systems or may be undesirable due to increased component costs. Software-based pacing avoids these concerns, albeit it is only a rate-control mechanism and other feedback mechanisms are needed to adjust the traffic rate. While QCN is suited for the data center where the feedback loop is short, large internet services now span multiple geographically-distributed data centers. In such scenarios software-based pacing at end hosts is desirable for flows traveling between data centers as higher-level protocols, such as TCP, and even the application itself will send traffic at a rate that matches the network capacity. Without software pacing, excess traffic must be stored either in TCP or socket buffers. While hosts have plentiful memory, this unnecessarily increases latency.

Token buckets [41, 6] are a widely used pacing model that has been implemented as the Token Bucket Filter (TBF) in the Linux kernel. While early versions of TBF were inaccurate at 100 Mbps due to course timer resolution, the advent of Linux's High Resolution Timers [3] has significantly improved performance at these speeds. However, verifying the performance of pacing tools at Gigabit speeds is difficult due to the large number of packets and small time scales involved. To address this problem, we introduce an efficient new software tool for measuring bandwidth and traffic burstiness at end hosts with microsecond granularity. We implemented our tool in the Linux kernel and use it to show the limitations of the Linux Token Bucket Filter (TBF) at speeds up to 10 Gbps.

We show that TBF works very well up to 1 Gbps. Despite this, we demonstrate that at speeds greater than 4 Gbps TBF has a number of issues. For example, at a specified pacing speed of 7 Gbps, TBF sends at nearly 10 Gbps, leading to a 40% overshoot. Even after fixing a bug that causes overshoot, we show that late timers in TBF can cause

undershoot of 25% at small bucket sizes. While this undershoot can be fixed by increasing the token bucket's size, this leads to additional traffic burstiness, buffering in the network, and the further delays latency-sensitive traffic. We show that this trade-off between switch-buffer occupancy and undershoot in average bandwidth is caused by late timers and explore some simple ideas to improve this trade-off.

This chapter is about making software pacing work in an emerging world of 10 Gbps links. In this context, our contributions are as follows:

- *Measurement tools to determine the effectiveness of Pacing:* The timer inaccuracy that plagues software pacing also affects the accuracy of measurement software. We work around this dilemma by a simple recalibration trick and describe simple tools based on this idea (Section 4.2) that we have implemented in the Linux kernel to measure bandwidth and burstiness at fine time scales.

- *Experiments to determine the effectiveness of Linux Software Pacing (TBF):* In Section 4.3, we describe simple experiments to evaluate Linux software pacing at speeds over 1Gbps and demonstrate its performance in the presence of system load.

- *Improvements to TBF:* In Section 4.4 we propose some simple changes to TBF to remove a prominent bug and to improve the trade-off between burstiness and achieved bandwidth.

The rest of the chapter is organized as follows. We describe the issues with software pacing via a simple model in Section 4.1 and describe our new tools in Section 4.2. We show our measurements in Section 4.3 and provide suggestions for improving TBF in Section 4.4. We describe related work in Section 4.5 and conclude in Section 4.6.

4.1 Software Token Buckets and their Problems

In the token bucket model of pacing, a bucket storing up to B tokens is continuously filled at a rate R. Before sending a packet of size P bytes, the controller checks that the bucket contains at least P tokens. If at least P tokens have accumulated in the

bucket, then P tokens are removed and the packet is sent. If there are less than P tokens in the bucket, then the packet is delayed until P tokens have accumulated. Since the bucket is filled at rate R, the bandwidth passing through the token bucket cannot exceed R bytes per second. However, the token bucket may transmit a burst of up to B bytes at line rate. Thus, the parameters of the token bucket model limit both the average rate and the maximum burst size of the resulting traffic stream.

In practice, it is inefficient for token bucket implementations to simulate a continuous stream of tokens filling the bucket. Hence one of two approaches is used to approximate this behavior. Hardware implementations often set a timer to fire at some fixed interval, T. At each timer event, RT tokens are added to the bucket and packets are sent if sufficient tokens have accumulated. In the second model, implemented in the Linux Token Bucket Filter, when each packet arrives at the token bucket, the number of tokens in the bucket is updated to account for the time elapsed since the last event. If sufficient tokens are available, the packet is immediately sent. Otherwise, a timer is set to fire at the time when sufficient tokens will have accumulated to allow the packet to be sent. When the timer fires, the number of tokens in the bucket is updated and the packet is sent.

Figure 4.1 abstracts the code implemented in the Linux Token bucket filter. The bulk of the work is done by a routine `serviceQueue` that fills the bucket based on the time elapsed since the last packet sent. The routine then tries to service as many packets as possible such that the total size of the packets serviced does not exceed the number of tokens in the bucket. Finally, if there are packets remaining in the queue, a timer is set to expire when there will be sufficient tokens to send the packet at the head of the queue.

Figure 4.2 shows how inaccurate timers can hurt performance of software pacing by reducing throughput below the specified rate because of "lost tokens". Assume that the last send event left the bucket with I tokens remaining. In an ideal model, the bucket will fill to capacity at time $\frac{B-I}{R}$. If the lag L before the next event or timer fire is greater than $\frac{B-I}{R}$, then $LR - (B - I)$ tokens will be discarded because the bucket size cannot exceed B and no packets will be sent during the lag period because the thread is asleep.

A token bucket pacer with perfect timers can also discard arriving tokens if there

```
B: maximum bucket size, parameter
R: fill rate, parameter
Q: queue of packets
lastTime: time last packet was sent
bucket: token bucket counter

Initially bucket = 0; lastTime = 0;

on_send_event(packet P)
  Place packet P in Q;
  serviceQueue();

on_timer_tick()
  serviceQueue();

serviceQueue()
  packet P = Head of Q;
  /* dequeue loop */
  while (P is not nil); do
    var now = getCurrentTime();
    var tokens = (now - lastTime) * R;
    tokens = Max (B, tokens + bucket);

    if (size(P) <= tokens ); do
      bucket = tokens - size(P);
      Remove (P) from Q;
      Send (P);
      lastTime = currentTime;
      P = Head of Q;
    else
      SetTimer (now + (size(P)-tokens)/R);
      return;
  done
end
```

Figure 4.1: Pseudocode for the algorithm implemented by the Linux Token Bucket Filter.

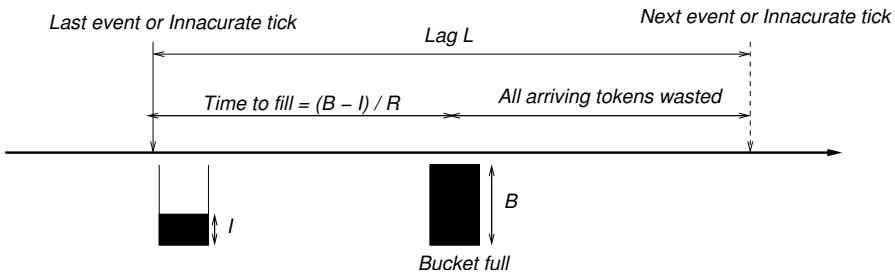


Figure 4.2: Bandwidth loss caused by inaccurate timers. If a timer expires after the buffer fills, all later-arriving tokens are wasted.

are *idle periods* with no data to send. However, tokens should never be discarded if there is a constant stream data to be sent. In reality, timers are imperfect and token discarding (and hence wasted send opportunities) can happen even when there is always data to send. For example, suppose the bucket size is B = 1 MTU and the rate is R = 1 MTU per time unit. Assume the inaccurate timer always fires after an interval $2/R$ instead of the desired $1/R$. Then the bucket will fill after the first half of the sleep interval, and the tokens accumulated during the remaining half will be wasted. Thus the inaccurate timer will cause the pacer to reduce its average throughput to $R/2$ instead of R even if there is an infinite amount of data to send.

More generally, suppose the distribution of lags is known and we plot the probability that the lag is greater than some specified time T (the complement of the CDF) as shown in Figure 4.3. Then whenever the lag exceeds $T = B/R$, the excess above B/R is wasted (assuming there is data to be sent). We can calculate the expected amount of wasted time as the area under the curve beyond B/R (shown shaded). To find the degradation in bandwidth, let $Y = Max(L - \frac{B}{R}, 0)$. We wish to consider the distribution of the random variable we call the *relative degradation* $D = Y/L$. Then the expected reduction in bandwidth is $E(D)$. For example, in the earlier example $D = 1/2$ always and hence the bandwidth is degraded by 50%.

Increasing the bucket size, B, reduces the probability that the inaccurate timer causes wasted tokens (this has the effect of shifting the threshold in Figure 4.3 to the right and hence reducing the degradation). However, doing so increases the burstiness of traffic and hence increases the buffer occupancies of downstream switches. Thus, inaccurate timers produce a trade-off between bandwidth and buffering, which ultimately

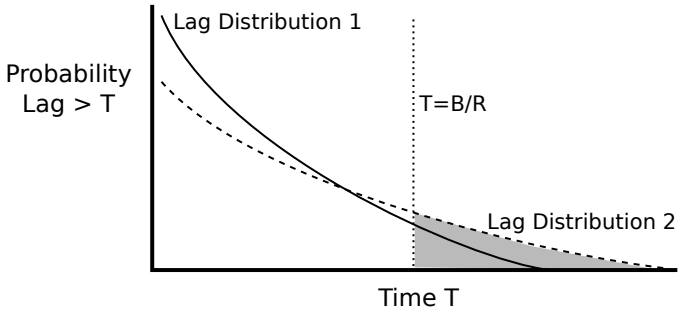


Figure 4.3: A probabilistic model for bandwidth degradation in the presence of inaccurate timers in software pacing. Lag distribution 1 is less wasteful because it has less area beyond the threshold.

affects latency. We will explore this trade-off experimentally in Section 4.3.3.

4.2 Tools for Measuring the Effectiveness of Pacing in Software

To evaluate a software pacing scheme we need tools to measure both bandwidth and "burstiness". For example, a simple tool may measure the peak bandwidth of a flow in any interval of say 10μs. This can be done simply by a counter, a timer that fires every 10μs, and a variable that holds the maximum observed value of the counter. However, we have just seen that the trade-off in software pacing is caused by inaccurate timers. This same inaccuracy will plague any *software* measurement tool operating at fine time scales.

The following simple idea, which we call *recalibration*, can be used to transform algorithms that work in a world of ideal timers to work with real timers. In particular, we will use recalibration to design software tools to measure the bandwidth achieved and buffering required by a flow.

The top of Figure 4.4 shows an ideal algorithm that applies some function to its state variables when either events (shown as E_1, E_2, etc.) or timer ticks (T_1, T_2, etc.) occur. In networking algorithms, the events are often packet send or receive events. The ideal algorithm assumes that the time between timer ticks is fixed at some ideal tick t.

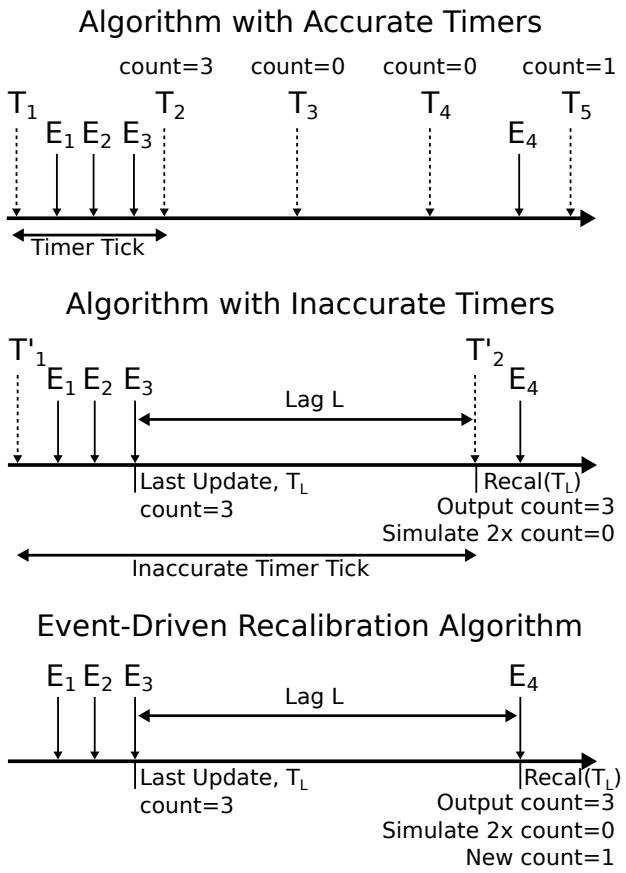


Figure 4.4: Illustrating Recalibration. A recalibration function is used to correct for missed timer ticks that should have occurred during the lag interval.

For example, to measure bandwidth in 10μs intervals, t is 10μs and the state variable used is a counter that is output and reset on each timer tick.

The middle figure shows the real situation. The events are unchanged but the timer ticks unpredictably, as denoted by ticks T_1' and T_2'. In the example, the inaccurate timer has missed 3 ideal timer ticks (T_2, T_3, T_4) but no events. To define the recalibrated version we need to specify a recalibration function $Recal$ that must fix-up the ideal algorithm's state to make up for any time it has missed. The recalibration algorithm utilizes a variable T_L that records the time of the last event or timer tick. In some cases, given a recalibration function, one may not need explicit timer ticks, but can simply call the recalibration function at events as shown on the bottom figure of Figure 4.4.

For example, if the ideal algorithm is a measurement algorithm using measurement intervals, $Recal(T_L)$ must calculate how many measurement intervals are missed

and adjust its output accordingly. While the simplest approach is to simulate the ideal algorithm output on each interval, in practice the real algorithm may miss 1000's of ideal intervals. Thus, it is important to define the outputs of the ideal algorithm carefully so that $Recal(T_L)$ can summarize the work of 1000's of intervals efficiently.

4.2.1 BandwidthTracker

Our first tool for measuring software pacing is BandwidthTracker, which efficiently samples bandwidth at microsecond resolutions and reports aggregate statistics (max, average, standard deviation). As input, it takes a traffic stream (either as a live stream or a log), a specified granularity G (e.g., 10μs), and a measurement interval M (e.g., 1 second). It outputs the maximum, average, and standard deviation of the bytes sent by the stream in every G units of time.

As discussed above, the algorithm for BandwidthTracker in a world with perfect timers is trivial. However, for small granularities, such as 12μs (the time to transmit 1500 bytes at 1 Gbps), the overhead of the basic algorithm is high due to constantly firing 12μs timers and error-prone due to the occasional inaccuracies that accompany short timers. Instead, an event-driven recalibration can be employed.

As shown at the bottom of Figure 4.4, an event-driven recalibration algorithm piggy-backs solely on events without explicitly setting timers. We give the pseudocode for our recalibrated BandwidthTracker in Figure 4.5. In our example, assume that each event E_i is a packet containing a single byte. At each invocation, BandwidthTracker recalls the timestamp of the previous event, T_L, and reads the current time, T. If $\lfloor T_L/G \rfloor = \lfloor T/G \rfloor$ then the current event occurred in the same sampling interval as the previous one, and the count is incremented. In our example, E_1, E_2, E_3 all occur during the first interval, so the counter is updated to 3.

If a new event does not occur in the same sampling interval, then BandwidthTracker simulates the ideal time tick for the end of the interval containing T_L by recording the value of the counter, then simulates any intervals with no events. In general, the algorithm needs to simulate $\lfloor T/G \rfloor - \lfloor T_L/G \rfloor - 1$ intervals with a counter of 0.

For the statistics tracked by BandwidthTracker we record the old count, then increment the number of intervals by $\lfloor T/G \rfloor - \lfloor T_L/G \rfloor$. Finally, BandwidthTracker

```
T_L: last time updated
T: current time
count: byte count for current interval
sum: total bytes across intervals
sum2: square of bytes for all intervals
ivals: total number of complete intervals

on_send_event(packet P):
  elapsed = floor(T/G) - floor(T_L/G);
  /* recalibration code */
  if (elapsed > 0); do
    sum = sum + count
    sum2 = sum2 + count^2
    ivals = ivals + elapsed
    count = sizeof(P)
  else
    count = count + sizeof(P)
```

Figure 4.5: Recalibration Pseudocode for the BandwidthTracker algorithm.

resets the counter and records the number of bytes of the packet arriving at T. In our example, when $Recal(T_L)$ is called at time T, the counter of 3 is recorded, zeros are recorded for the next $(T_4 - T_1 - 1) = 2$ intervals, and the counter is set to 1 to record the progress so far in the fourth interval.

BandwidthTracker works particularly well in scenarios where events may be separated by long delays, as the cost of running $Recal(T_L)$ is always O(1). We can further optimize the speed of our event-driven recalibration algorithms by restricting the granularity of G to powers of 2. This allows us to replace the division and floor functions with simple right bit-shift operations. Such optimizations are of particular significance for hardware implementations.

Note that our algorithm bears some resemblance to Partridge and Garrett's TB algorithm [41] and to subsequent generalizations by Tang and Tai [45]. However, the

goal of TB is to take a time-stamped stream of packets and to find a set of token bucket parameters that are consistent with that stream. In other words, find a token bucket with a set of parameters such that all packets within the stream will be sent out immediately. This is particularly useful for measurement-based admission control.

However, our goal here is not to find a set of token bucket parameters consistent with a stream but to simply implement any reasonable measure(s) of burstiness of a stream. BandwidthTracker calculates the average and variation of the bandwidth in any specified interval and could be used in conjunction with more complex aggregation techniques [46]. Our next tool, BufferSim, takes a different tack as it simulates the buffer size at a next hop queue as a measure of burstiness. Thus our algorithms are simpler and different from that of [45].

4.2.2 BufferSim

BandwidthTracker provides some indication of the "smoothness" of a flow at any specified time granularity and can be used for debugging a wide range of problems. However, to investigate the impact of queueing on latency, we now focus on tracking buffer occupancy. The classical way to formally extract buffer sizing information from traffic models is by queueing theory. However, the use of queueing theory is daunting because modeling the distribution of the lag (as in Figure 4.3) from machine to machine is hard, and because solving the resulting G/D/1 queueing system may be hard.

To bypass this dilemma, we introduce a second tool called BufferSim. BufferSim emulates buffers of any specified size in real-time when supplied with a model of how the buffer is drained and a live traffic stream. In other words, BufferSim provides an online simulation of the occupancy of an imaginary buffer. BufferSim allows an analyst or algorithm to experiment with "what-if scenarios" (e.g., how does the latency or drop rate vary for an HDTV stream if the buffer size is doubled or the switch uses cut-through) without physically implementing the buffer.

BufferSim takes as input an input traffic stream S, a buffer size B in cells, a cell size C, a specified granularity G (e.g., 10μs), and a measurement interval M (e.g., 1 second). It outputs the number of dropped packets that would occur due to insufficient buffer capacity. In addition, BufferSim also simulates an unbounded buffer and report

```
C: bytes per cell,  integer
max: maximum buffer size seen
drops: simulates number of drops

on_send_event(packet P):
  cells = ceiling (size(P)/C);
  U = U + cells;
  max = Max (U, max);
  If F + cells <= B then
    F = F + cells;
  Else drops = drops + 1;

on_timer_tick:
  U = Max(0, U - 1);
  F = Max(0, F - 1);
```

Figure 4.6: Pseudocode for the Ideal Buffer Simulation.

the maximum buffer occupancy produced by the traffic stream.

The ideal algorithm for BufferSim is shown in Figure 4.6. This algorithm sets a timer to fire at the specified granularity and keeps two byte counters: F simulates a finite buffer of maximum size B and U simulates an unbounded buffer. At each timer event, if F or U is non-zero, then it is decremented by one cell. For each packet event, we check if there are sufficient cells to add the packet's size to F with out exceeding B. If F would exceed B, then F is not increased and $drops$ is incremented. However, U is always increased by the number of cells required to store the packet.

The recalibrated version is shown in Figure 4.7. Variable T_L records the last update time (last packet received). The simple insight is that even if an arbitrary amount of time has passed since the last packet reception, it cannot affect either the max buffer size or number of drops. However, we need to simulate the buffer drain model over the missed time by decrementing F and U by the number of cells that could have been dequeued in this period, taking care to leave these variables non-negative.

```
T_L: last time updated
T: current time
G: granularity
Recal():
  elapsed = floor(T/G) - floor(T_L/G);
  F = Max(0, F - elapsed);
  U = Max(0, U - elapsed)
```

Figure 4.7: Recalibration Pseudocode for the BufferSim algorithm.

4.2.3 Linux Kernel Implementation of Tools

We have implemented BandwidthTracker and BufferSim in the Linux 2.6.32.8 kernel. In order to observe incoming and outgoing packets, we inserted code between the NIC driver and the IP layer in the networking stack, as shown in Figure 4.8. For each socket buffer, we enable nanosecond-resolution timestamping, which records the system time for incoming packets as the device driver receives them. For outgoing packets, we note the system time and update our statistics after the traffic has been shaped by TBF, but before it passes into the driver.

Monitoring is enabled by specifying a source/destination IP-address pair, a sampling interval in nanoseconds, and the number of samples to aggregate together when reporting statistics. The aggregate statistics from BandwidthTracker and BufferSim are accessible to user-space programs through a system call interface. Our user-space logger polls once per aggregation period to receive the aggregated data and write it to a log file. Future programs could use this interface to actively monitor their behavior and dynamically tune themselves.

While the bulk of this chapter assumes a software setting, we believe it is useful for routers and other hardware devices to deploy tools such as BandwidthTracker and BufferSim to determine the smoothness of flows and their impact on buffer occupancy. The tools are sufficiently simple to be easily implementable in hardware ASICs. While recalibration is strictly not necessary in a hardware setting, it allows the use of a coarse timer for recording data each aggregation period, which should simplify implementation

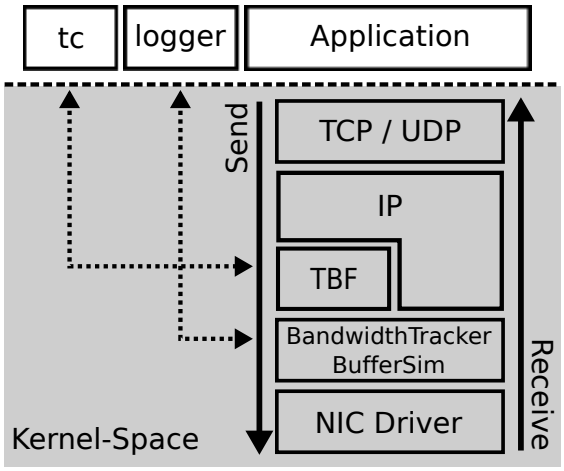


Figure 4.8: Software Organization. BandwidthTracker and BufferSim are implemented below the IP layer and Token Bucket Filter. The `tc` utility enables and configures TBF, while our custom user-space logger configures and records data from the BandwidthTracker and BufferSim.

and reduce power.

4.3 Evaluating Linux Token Bucket Filter

In our experiments, we used two 2.3Ghz, hyperthreaded, quad-core, Intel Xeon servers, each with 24GB of RAM and two 10Gbps Myricom network interfaces. Our machines are running our instrumented version of the Linux 2.6.32.8 operating system. We connected these two machines through a 10 Gbps Cisco Nexus 5020 switch. Except where noted otherwise, our traffic workload is generated by a custom UDP sender application that spawns four threads that wait in a tight loop sending UDP packets as fast as possible. We used the `tc` utility to configure the Linux TBF to rate-limit outbound traffic at various speeds.

We chose to use a custom UDP application in place of *iperf* [5] because we could vary more parameters such as simulating distributions of packet sizes. We needed to use 4 sending threads since our initial experiments indicated that this setup achieved the highest number of send events per second (1.3 million) with the fewest number of threads. Using 4 threads, we could reach transmit speeds just below 10 Gbps using

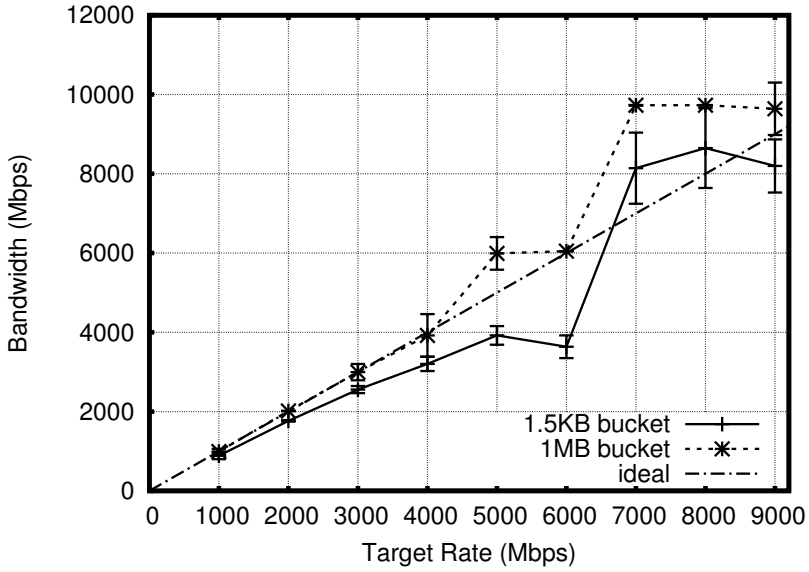


Figure 4.9: Observed bandwidth deviates from the specified rate. With a large 1MB bucket the observed rate overshoots the target rate due to insufficient precision when calculating inter-packet spacing. For a small, 1 MTU bucket size, late timers cause undershoot at values below 6 Gbps but precision errors cause overshoot at 7 and 8 Gbps.

1500-byte packets. Burstiness calculations were done using BufferSim with a drain model that uses 64 byte cells – the same cell size used in our Cisco switches. Average bandwidth is measured using BandwidthTracker with a granularity G of 64μs. Error bars on our graphs show the standard deviations across 60 separate runs of 1 second each.

4.3.1 Bandwidth Problems with TBF

We begin by demonstrating the performance of the Linux Token Bucket Filter at target rates above 1Gbps. Figure 4.9 shows the achieved average bandwidth for given bucket fill rates for a very small bucket (1 MTU) and at a reasonably large bucket (1 MB). The ideal bandwidth is shown as a straight line of slope 1.

The 1MB token bucket performs as expected up to 4 Gbps. After this point it frequently overshoots the target bandwidth. For example when the target bandwidth is 5 Gbps and the bucket size is 1 MB, the average measured bandwidth was close to 6

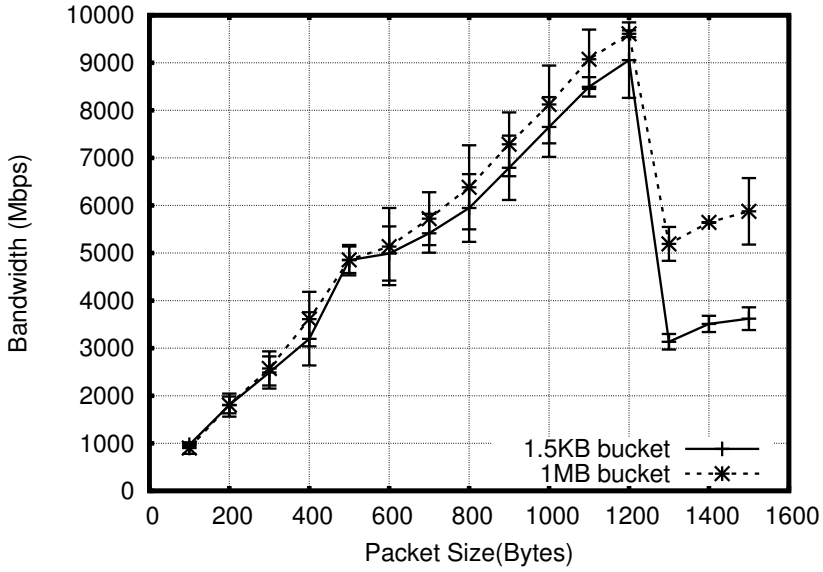


Figure 4.10: TBF is run with a target rate of 5 Gbps. At packet sizes less than 500 Bytes the bandwidth is constrained due to speed of send system calls. For packet sizes greater than 500B, insufficient precision in inter-packet spacing cause overshoot.

Gbps, an overshoot of 20% with a standard deviation of 0.4 Gbps over 60 runs. The situation worsens considerably at higher target rates. For example, at a target rate of 7 Gbps, the measured average is 9.7 Gbps (essentially line rate) with a standard deviation of 0.06 Gbps.

On the other hand, the 1.5 Kbyte token bucket undershoots the target bandwidth beyond 2 Gbps. At a target of 5 Gbps, the average throughput is 3.9 Gbps, an undershoot of 22% with a standard deviation of 6%. At 6 Gbps, the average is even lower at 3.64 Gbps, a 25% undershoot, while at 7 Gbps, there is an overshoot of 12%.

To further investigate this behavior, we configured the TBF with a fixed rate of 5 Gbps and varied the size of our UDP packets. The results are plotted in Figure 4.10. Note that the ideal (not shown) is a horizontal line at 5 Gbps. At packet sizes less than 500 bytes, the sending rate cannot be achieved because of the overhead of system calls. However, beyond this point the achieved bandwidth continues to increase for both bucket sizes, reaching 9.6 Gbps at 1200 bytes per packet and a 1 MB bucket – an overshoot of 92%!

We investigated this phenomenon and found a bug in the `tc` utility used to set the

token bucket's kernel parameters. First, observe that the token bucket code in Figure 4.1 requires a division to set timers for arbitrary target rates. However, Kernel code does not to support floating point arithmetic, thus the Linux Token Bucket Filter uses a lookup table that maps the shortfall in tokens required to send the next packet to the delay required for those tokens to accumulate. This lookup table is computed in user-space by `tc` and passed into the kernel when the token bucket is initialized. Due to integer-type variables used in `tc`, times that are computed with microsecond resolution have their fractional components discarded.

For example, in the ideal scenario, a 1500-byte packet should be sent every 2.4μs to achieve a rate of 5 Gbps. However, due to the truncation of partial microseconds in `tc`, the value stored in the lookup table is 2μs. Transmitting full MTU packets at an inter-packet spacing of 2μs results in a bandwidth of 6 Gbps, which agrees with our observed value in Figure 4.10. Because of this problem, all packets with size less than 1250B are sent at 1μs intervals, and packets between 1250B and 1500B sent every 2μs.

At rates below 1 Gbps, this problem does not manifest itself as the disregarded nanoseconds have minimal impact on inter-packet delays of 10's to 1000's of microseconds. However, as we have seen here, commonly used tools should be re-evaluated as bandwidths scale to multiple gigabits per second.

4.3.2 Average Bandwidth after Fixing `tc`

We corrected this bug in the `tc` utility by ensuring that all variables used in computing the TBF lookup table were at nano-second resolution. To validate our fix, we re-ran the previous two experiments. These results appear in Figures 4.11 and 4.12. Note that the large bucket experiments now track much more closely to the specified target rates. For example, at a target of 6 Gbps, the average is 5.7 Gbps with a standard deviation of 0.7 Gbps; at a target of 7 Gbps, the average is 6.9 Gbps with almost no variance.

However, at small bucket sizes, there is still significant undershoot. For example, at a specified bandwidth of 6 Gbps, the average is only 3.6 Gbps, which is a 40% undershoot with a standard deviation of 0.3 Gbps. We will show that this undershoot is caused by late timers that cause tokens to be dropped as in Figure 4.2.

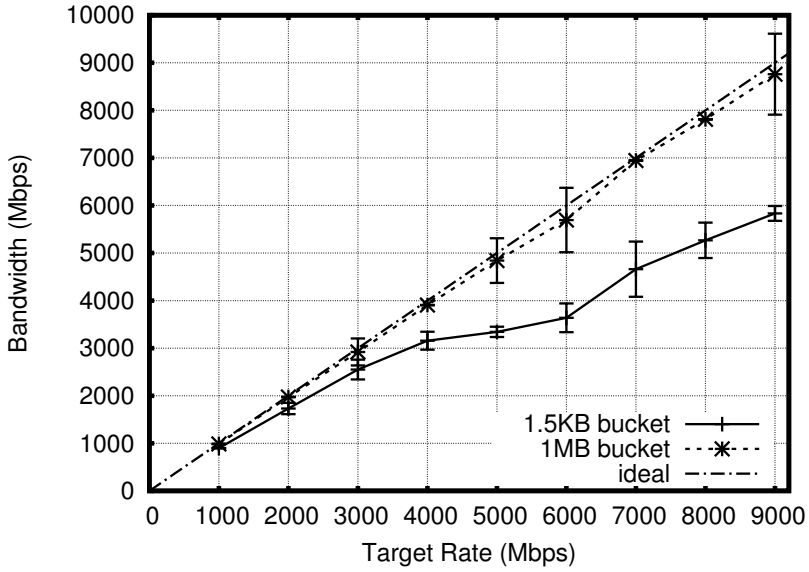


Figure 4.11: Specified versus observed bandwidth after fixing precision bug. Experiments run with 1500-byte packets.

Some initial evidence that this undershoot is caused by late timers can be seen from Figure 4.12 at a fixed target of 5 Gbps. Note that once again the average bandwidth at low packet sizes is less than 5 Gbps because of the overhead of system calls. However, after 600 bytes the undershoot increases with packet size.

It is clear to see the cause of this undershoot by considering the following example. Consider a packet size of 750 bytes and a bucket size of 1500 bytes. At 5 Gbps, sending a 750-byte packet causes a timer to be set for 1.2μs. Even if this timer is late by 1.2μs (such lateness is rare in our system, as we show later), tokens will not be lost because they "fit" into the bucket size of 1500 bytes. On the other hand, when sending a 1500-byte packet, the timer is set for 2.4μs. If the timer is 1μs late, then 1μs worth of tokens are lost forever. In such a case, any time inaccuracy causes missed sending opportunities. Thus, the smaller the packet size relative to the bucket size, the less likely it is for tokens to be lost.

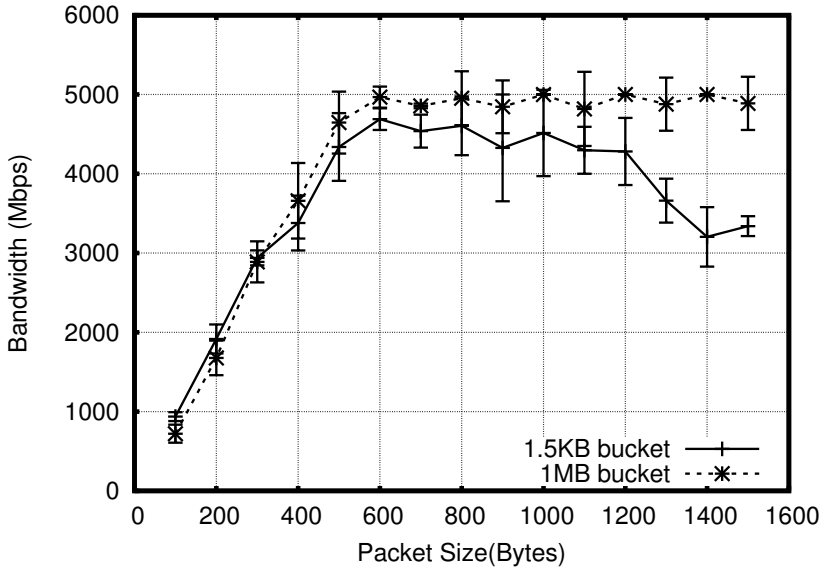


Figure 4.12: Average bandwidth versus packet sizes after correcting the precision bug for a fixed target rate of 5Gbps.

4.3.3 Balancing Bandwidth and Buffering

If undershoot is due to lost tokens as in Figure 4.3, then the probability of lost tokens can be lowered by increasing the bucket size because it moves the vertical line T to the right. However, this will also increase the burstiness of TBF. Figure 4.13 shows this trade-off by plotting both the average bandwidth (solid line) and the maximum "buffer" occupancy measured by BufferSim as bucket size increases. The target is fixed at 5 Gbps.

First, note that at small bucket sizes, such as 1.5 Kbytes, the average bandwidth is only 3.1 Gbps, an undershoot of 38%. At a bucket size of 2 Kbytes, the average bandwidth is 4.4 Gbps, reaching 4.9 Gbps at 3 Kbytes. However, the simulated maximum buffer sizes to required to absorb the corresponding bursts generated at these bucket sizes are 2675, 3080, and 4115, respectively. While this does not seem like a big increase if regarded as an additive increase of 1.5 Kbytes, it is more serious when viewed as a *multiplicative increase of 54%* in queueing delay.

Further, late timers are likely to be worse as the system load increases. The TBF code is driven by timer interrupts tied to a single core. In our multicore environment

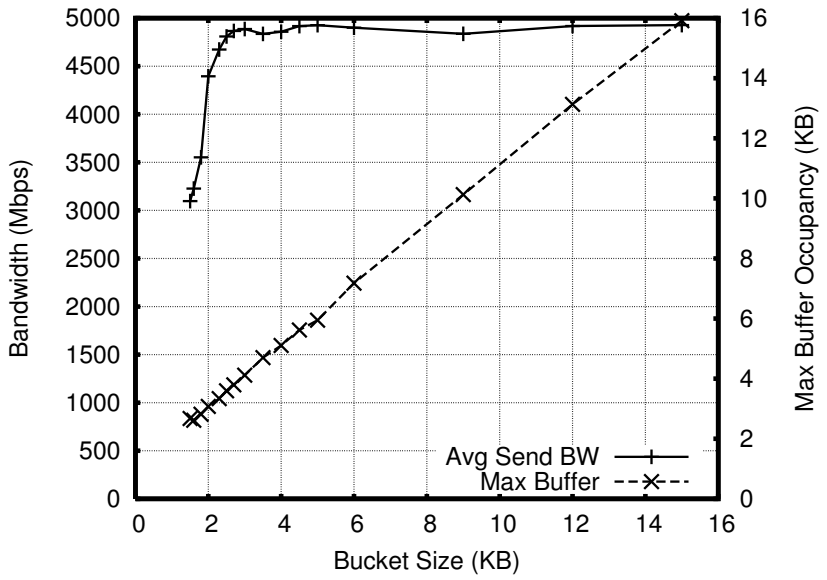


Figure 4.13: The average bandwidth and maximum buffer occupancy with a target rate of 5Gbps for various bucket sizes.

TBF does not incur significant overhead – consuming only one of the 8 cores in our machine. However, TBF performance may be affected as contention for interrupt handling increases. To investigate this hypothesis, we subjected the sending host to additional load caused by a high volume of received UDP traffic (as fast as it could be sent from another host) in addition to the 4 threads trying to send UDP traffic. Figure 4.14 shows the average bandwidth achieved (dotted line) with interference. While not shown, the maximum buffer occupancies with and without interference are virtually identical. We see that even at bucket sizes of 6 and 9 Kbytes, the average bandwidth with interference does not exceed 4.5 Gbps. This suggests that a much higher multiplicative factor in buffering at high loads, even up to a factor of 4 or 5.

If a large number of hosts connected to a commodity switch with a small amount of buffers are doing software pacing, such an increase can significantly impact latency. Assuming a 48 port switch, and two 5 Gbps rate limited flows per port and a multiplicative factor of 5 to account for high load, we could expect a worst case buffer occupancy of 600 Kbytes just to cover for the inaccuracy of software timers. While this extra buffer occupancy increases the chances of dropped packets, its more serious problem is the increase in latency. This cost will worsen at higher speeds as timer accuracy diminishes

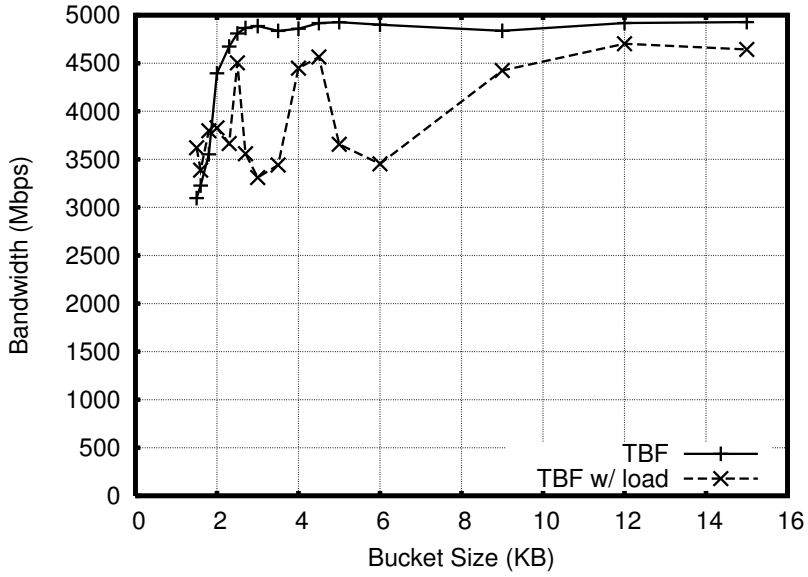


Figure 4.14: Variability of observed bandwidth for various bucket sizes in the face of interference from high volume of UDP receive traffic.

for short intervals.

Chained Token Bucket Limitations A frequently advocated approach to using large bucket sizes while limiting the affects of bursts is to use two token buckets chained serially. In this model, the rate of the first bucket, R_1 is set to the desired average bandwidth and given a bucket size B_1 that is sufficiently large as to mitigate the effects of late timers (or gaps in the packet stream). The second bucket is configured to smooth any bursts of traffic released by the first bucket by setting R_2 to a "peakrate" greater than R_1, and B_2 to the size of 1 MTU. Thus, any bursts released by first bucket will be paced out at R_2 instead of the line rate.

The Linux TBF implements this feature, which can be enabled through `tc`'s `peakrate` option. We show its performance in Figure 4.15. For this experiment we set a target rate of $R_1 = 5$Gbps, a bucket size of $B_1 = 1$Mbyte, and varied the peakrate, R_2. The chained token bucket is able to reclaim some of lost sending opportunities. For example at a peak rate of 8Gbps, the observed bandwidth is 4.9Gbps, but the maximum buffer occupancy is 880 Kbytes – two orders of magnitude more than the standard TBF. Since the chained bucket is implemented with the same inaccurate timers it suffers from

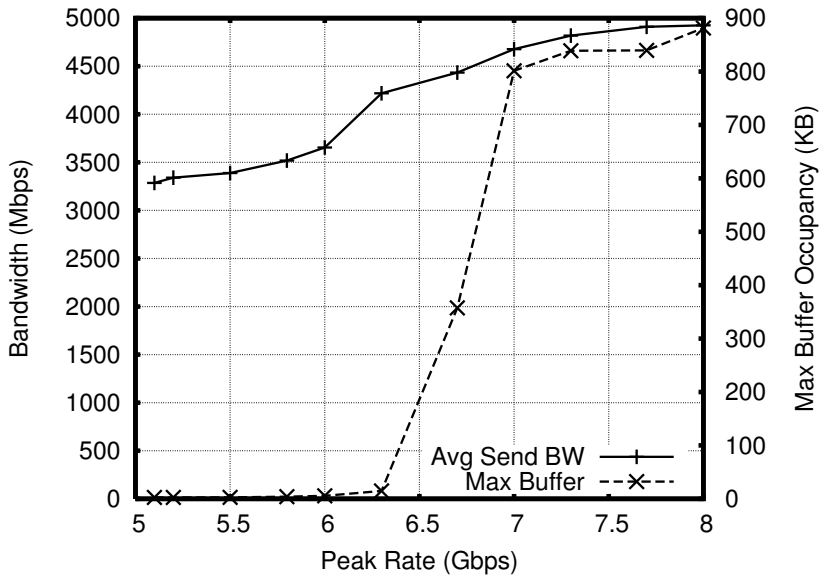


Figure 4.15: Performance of chained token buckets. The average bandwidth and maximum buffer occupancy with a target rate of 5Gbps and a bucket size of 1 MByte are shown for varying peak rates.

the same problems as the standard token bucket. However, running the buckets in series compounds these problems.

4.3.4 Dissecting Timer Settings in TBF

To verify our hypothesis that the trade-off is caused by lost tokens, we instrumented the Linux kernel with a counter that tracks lost time. For example, if the bucket size is 1500 bytes and the target is 5 Gbps, when the timer fires at 3.4μs instead of 2.4μs, we add 1μs to the counter. Figure 4.16 shows the percentage of time lost as bucket size varies. At a bucket size of 1.5 Kbytes, we see that the percentage waste is 39% and at 2 Kbytes it is 12% which is consistent with the results in Figure 4.13, thus confirming our hypothesis.

Timer Distributions Next we investigated the distribution of timers set by TBF after fixing the precision bug. For example, at a token bucket size of 1500 bytes and a target rate of 5 Gbps, we might expect the vast majority of timers to be set for 2.4μs. However,

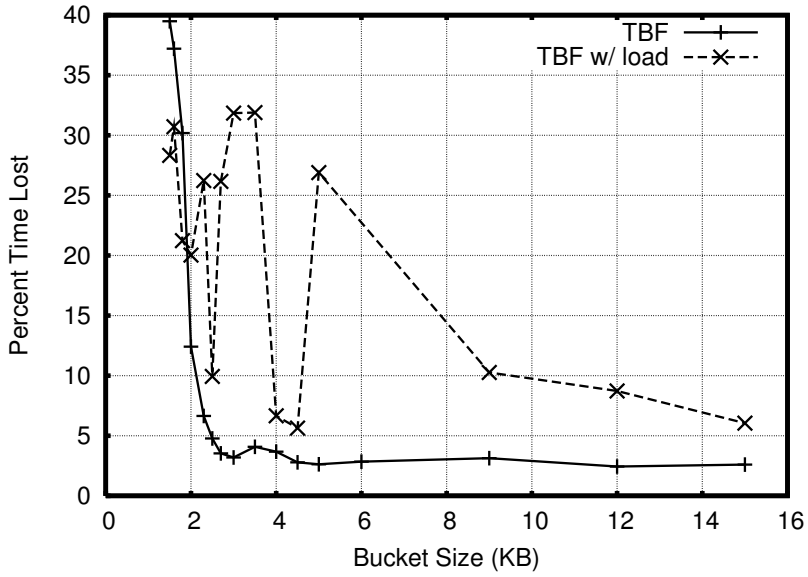


Figure 4.16: Percentage of time lost due to overflowing token bucket versus the size of the token bucket. Shown with and without load for a target rate of 5Gbps and a packet size of 1500 bytes.

we found that timers were set according to the distributions in Figure 4.17, shown in the case of low load and high load.

The most surprising phenomenon was the presence of negative and zero timers. Notice that without load 12% of the timers were set to at or below zero. Further, with high load 17% of timers were set with target values in the past.

There are several factors contributing to negative and zero timers. First, the code often sets small timers (around 68% of timers in Figure 4.17 are set for less than 500nsec). This happens because TBF sets the timer value on every event, even when the timer is already set. For example, suppose that the first packet is due to be sent in 2.4μs and a timer is set to expire at that time in the future. If a second packet arrives at the token bucket 2.0μs later, then TBF will check the current time and compute that the first packet is due to dequeue in another 0.4μs. Since only one timer is maintained, TBF resets the timer for this time, overwriting the original expiration time with the same value.

Such small timer values can also lead to negative timers for the following reason. The timer expiration is specified in absolute time and is calculated based a timestamp

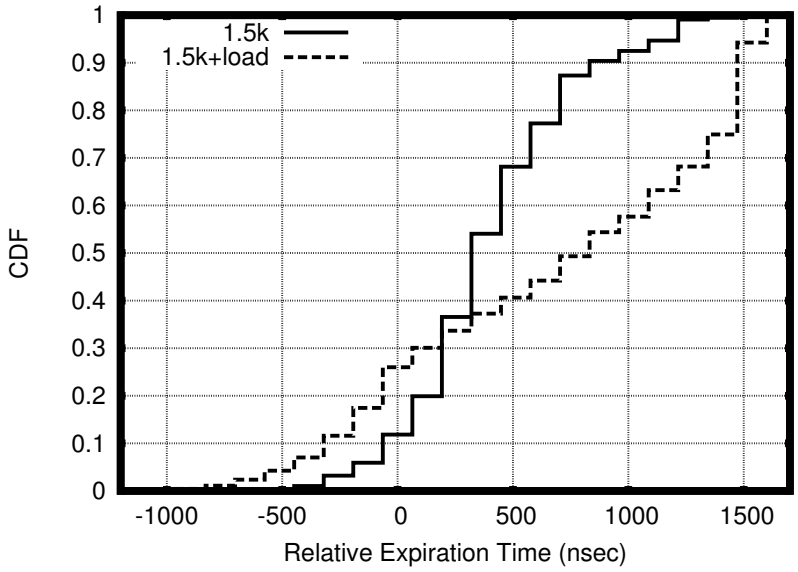


Figure 4.17: For all timer fires, the distribution of target times set with and without interfering UDP receive load. Collect from TBF at a target rate of 5Gbps & bucket size of 1500 bytes.

taken at the start of the dequeue loop. In the time required to calculate the timer's expiration time and to call the timer routine, sufficient time can pass (to execute the code) such that the expiration time is now in the past. Since we measure the relative expiration time for the point immediately after the timer is set, this produces a negative value. The reason the time to execute the code is variable is because the timer must acquire a lock to schedule itself. If there is contention for the lock (likely at high loads), then this time can increase, which explains the occasionally large values of negative timers, even up to -1.8μs.

Intuitively, spuriously readjusting timers could increase the number of timers set to be significantly greater than the packets sent. We measured this and found this to be true. For example, at 5 Gbps target and 1.5 Kbyte bucket size, during a 60 second period, the code enqueued 15.6 million packets. However, the number of timers set or readjusted was 28.2 million, almost double the number packets. The number of timers that actually fired was 14.9 million. While this is 700K less than the number of enqueued packets, we note that events caused by packet enqueues also update the state of TBF and allow packets to pass through without waiting for a timer to fire if sufficient tokens are

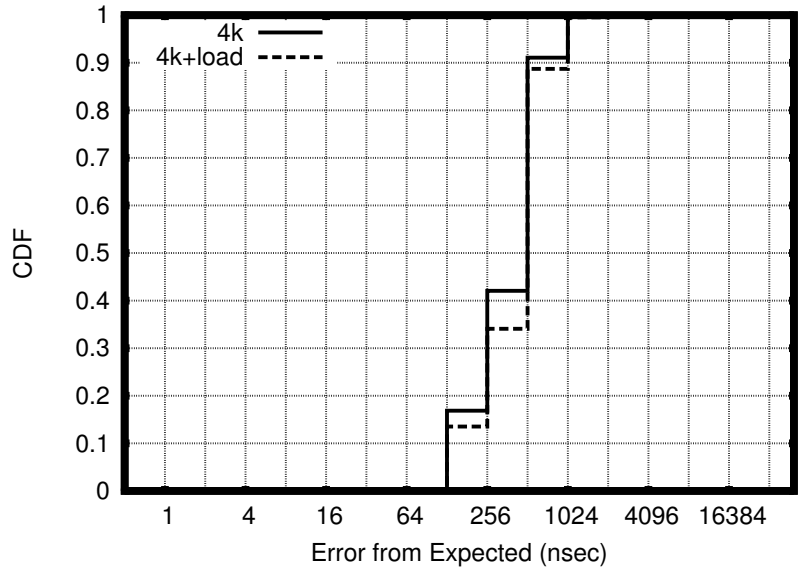


Figure 4.18: Distribution of timer error (time between desired expiration time and actual timer callback) for a target of 5Gbps and a bucket size of 4 Kbytes, with and without interfering UDP receive load.

available. At the very least readjusting the timer is unnecessarily baroque; at the very worst, it increases the contention for timers and results in more late timer fires.

Timer Error Next, we measure the error in timer values. In other words, given that a timer has been set for time T and fires at a later time T', the error is $T' - T$. We show the distribution of errors at a target rate of 5 Gbps and a bucket size of 4K with and without added load. Figure 4.18 shows, for example, that 42% of the timer errors is less than 512nsec without load, while only 34% of timers achieve this accuracy with load. However, we see that regardless of load approximately 90% of timers are no more than 1μs late. While not shown on this graph, 0.02% of the timer events are late by 16μs or more, with some exceeding 65μs. We see that the addition of load from UDP receive events affects the lateness timers, as well as, the number of timers set in the past.

Finally, we investigated the relationship between when a timer is set to expire and the likelihood that it will be fire on time. In Figure 4.19 we show the distribution of observed timer error for short intervals (between 128 and 256nsec) and long interval (between 1408 and 1536nsec). We see that longer timers fire less than 256nsec late

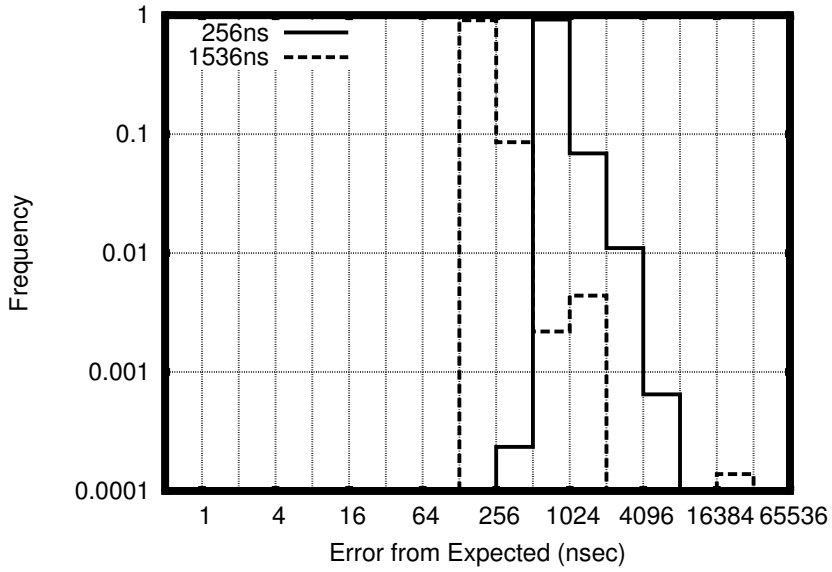


Figure 4.19: Distribution of timer error for long and short timer expirations. Short timers (128 to 256nsec) are between 512 and 1024nsec late with 92% probability, while longer timers (1408 to 1536nsec) fire within 256nsec of the target 91% of the time.

with 91% probability, while short timers are frequently 512 to 1024nsec late. This is particularly troubling for the TBF implementation since, as we noted above, events occurring while a timer is already set will reset the timer's value, often to a short (or even negative) interval, thus increasing the lateness for the timer event.

4.4 Improving the TBF code

Based on our measurements, we modified the Linux TBF code to address some of the inefficiencies we found.

4.4.1 Avoid Timer Readjustment

From the pseudocode in Figure 4.1, we see that each time a thread of execution enters the `serviceQueue` loop either a packet is sent or a timer is set. However, if the function attempts to set the timer while it is already active, the expiration time is updated with the new value. If `serviceQueue` is run several times before the next packet is

sent, then each call will readjust the timer for the same expiration time. This process of readjusting the timer adds additional overhead due to calls to timer subsystem functions, some of which must acquire locks. In addition, Figure 4.19 indicated that setting short timers increases the magnitude by which timers are late timer. Hence, resetting the timer is both redundant and detrimental to performance.

To quantify this problem we observed the number of times the timer is set or readjusted versus the number of timers that actually fire. We summarize this data in Table 4.1. During a 60 second run, 28.2 million timers are set, but only 14.9 million actually fire. This means that 47% of the calls to the time subsystem are unnecessary.

To address this inefficiency, we skip the timer readjustment if the timer is already active. We show the modified pseudocode for `serviceQueue` in Figure 4.20. As seen in Table 4.1, this simple change eliminated all spurious calls to the timer subsystem and resulted in a 15% improvement in bandwidth with no increase in the maximum buffering. Additionally, this modification did not adversely affect the bandwidth achieved at large bucket sizes.

In Figure 4.22 we show how this modification (labeled: TBF, no resets) affects the achieved sending bandwidth as the token bucket size varies. At bucket sizes less than 2 Kbytes the performance difference between TBF and our modified "no resets" code is most pronounced since the token bucket is not able to store extra tokens that arrive on a late timer event. Since we prevent timer readjustment, our modified code utilizes longer timers that have less error.

4.4.2 Send Early

From Figure 4.17, we observed that a large number of the scheduled timers are set to expire in the past or in the very near future ($<$500nsec). However, Figure 4.18 showed that these timers are frequently late by up to 1μs. These late timers result in lost sending opportunities if the bucket is small, or more bursts if the bucket is large. To address this problem, we modified that code to allow the token bucket to send early if the timer would be set for an interval less than a specified delta.

To ensure that we maintain the overall average bandwidth, we use up all available tokens, and store a negative token count in the bucket. This is very similar to

```
serviceQueue()
  packet P = Head of Q;
  /* dequeue loop */
  while (P is not nil); do
    var now = getCurrentTime();
    var tokens = (now - lastTime) * R;
    tokens = Max (B, tokens + bucket);

    if (size(P) < tokens ); do
      bucket = tokens - size(P);
      Remove (P) from Q;
      Send (P);
      lastTime = currentTime;
      P = Head of Q;
    else
      /* don't readjust if already set */
      if (not TimerActivated()); do
        SetTimer (now + (size(P)-tokens)/R);
      return;
  done
end
```

Figure 4.20: Updated pseudocode for `serviceQueue` from Figure 4.1 to avoid timer readjustment.

keeping a deficit counter as in Deficit Round Robin [47]. We give the pseudocode for our algorithm in Figure 4.21.

We show the results of this modification in Figure 4.22 and Table 4.1. We see that sending early increases the achieved bandwidth at 1500B to 4.4Gbps, an improvement of 41% over standard TBF, but also increases the maximum amount of buffering required from 2675 to 3135 bytes, an increase of 17%. Notice that the increase in buffering is due to the fact that packets can now be spaced more closely together. Additionally, our average bandwidth is increased both by avoiding short timers, and also by allowing a deficit of tokens in the bucket. This deficit allows the token bucket to gracefully handle some of the late timers. For example, suppose that our code is configured with a rate of 5Gbps, a bucket size of 1500 bytes, and delta of 1μs. If a packet is sent 1μs early, then the bucket will now contain -625 bytes and the next packet is scheduled to be sent at 3.4μs in the future. However, if other packet-send events cause `serviceQueue` to run any time between 2.4 and 3.4μs, another packet will also be sent early. Hence, allowing a negative token count effectively increases the range of our bucket and also avoids setting small, inaccurate timers.

4.5 Related Work

Timing wheels [47] were proposed in 1983 to allow efficient timer algorithms and were added to FreeBSD and later as part of Linux Jiffies. Aron and Druschel propose the use of SoftTimers [13], to provide a probabilistic timer facility that often (but not always) ticks in microsecond intervals. The key idea is to leverage common operating system events such as system calls and hardware interrupts (which incur the context switch overhead anyway) to check and fire timers. In 2006, High Resolution Timers [3] appeared in Linux leveraging improved hardware support to provide microsecond-accurate timers.

In addition to the Linux Token Bucket Filter, software pacing has been implemented in Linux's Hierarchical Token Bucket[22, 6] and in PSPacer[8]. While hierarchical token buckets is a separate code base that enables policies for pacing sets of flows, the underlying token bucket model is the same and has not been evaluated at high

```
delta: amount of time to allow early send

serviceQueue()
  packet P = Head of Q;
  /* dequeue loop */
  while (P is not nil); do
    var now = getCurrentTime();
    var tokens = (now - lastTime) * R;
    tokens = Max (B, tokens + bucket);

    if (size(P) < tokens + (delta*R)); do
      bucket = tokens - size(P);
      Remove (P) from Q;
      Send (P);
      lastTime = currentTime;
      P = Head of Q;
    else
      SetTimer (now + (size(P)-tokens)/R);
      return;
  done
end
```

Figure 4.21: Updated pseudocode for `serviceQueue` from Figure 4.1 for sending early to avoid setting short timers.

Table 4.1: Measurement of average achieved bandwidth, maximum buffer required, number of packets enqueued, number of calls to dequeue, the number of timers set/readjusted, and the total number timer fires for our two improvements - sending early (1024nsec delta) and eliminating timer resets.

Implementation	Avg. BW	Max Buffer	Enqueues	Dequeues	Timers Set	Timer Fired
TBF (original)	3096 Mbps	2675 Bytes	15.6 M	43.8 M	28.2 M	14.9 M
No timer resets	3553 Mbps	2675 Bytes	18.0 M	50.6 M	17.7 M	17.7 M
1024nsec delta	4376 Mbps	3135 Bytes	22.1 M	49.1M	27.0 M	11.9 M
1024nsec delta & no resets	4370 Mbps	3205 Bytes	21.5 M	50.5 M	16.4 M	16.4 M

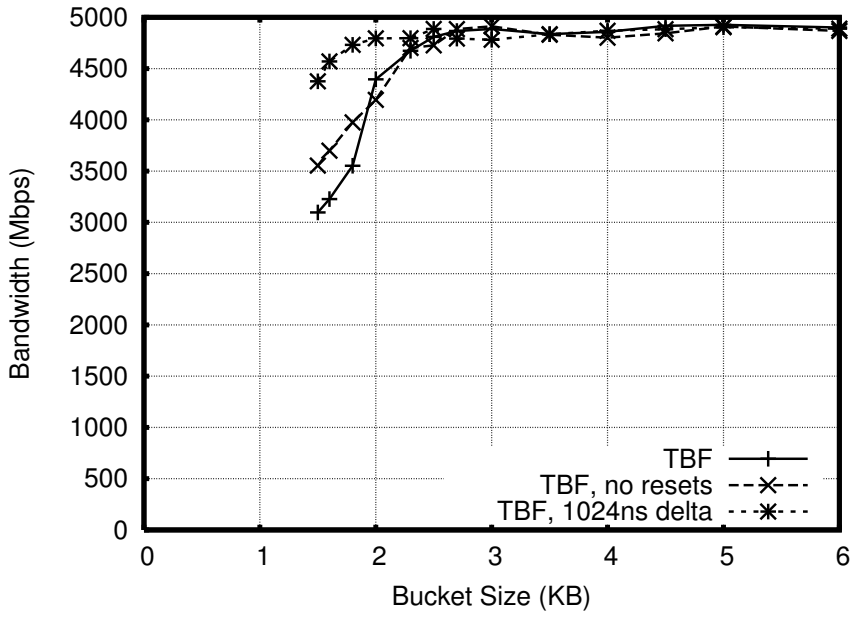


Figure 4.22: Bandwidth vs. bucket size for our two improvements - sending early (1024nsec delta) and eliminating timer resets.

speeds.

PSPacer is a software pacer for Linux (see also [44]) that adds artificial ”gap” packets (e.g. 802.3 PAUSE frames that are discarded by switches) to precisely control the inter-packet spacing between real packets. While this is an alternative to soft pacing as in TBF, there are some issues with [44]. First, the evaluation in [44] was done at 1 Gbps. While this was clearly superior to TBF in Linux when Linux was using millisecond timers, we have seen that TBF works well at 1 Gbps and only manifests issues at speeds over 4 Gbps. Thus PSPacer also needs evaluation at higher speeds. Further, PSPacer is based on hijacking PAUSE frames which makes them unusable for congestion control. The paper [44] justifies this by saying that receive congestion should rarely occur at 1 Gbps with modern PCs. However, this premise may not hold at 10 Gbps.

The evaluation of PSPacer makes use of a custom, hardware buffer simulator that implements a buffer drain model similar to the one we propose with BufferSim. However, we show that accurate buffer measurement can be implemented solely in software by utilizing our Recalibration Paradigm.

The Recalibration Paradigm is inherent in many works but, as far as we are

aware, is not clearly abstracted. For instance, SoftTimers can be considered an instance of this paradigm where the OS events recalibrate the required firing of timers. Many algorithms use an event-driven approach where packet sending events are used to recalibrate based on elapsed times but the essence of recalibration is often entangled with the specifics of the implementation.

Microbursts are a common term for short-lived bandwidth spikes that can cause significant latency increases for remote disk access [42] and many other data center applications [19]. NetScout and Corvil are hardware boxes that provide microburst alarms. Vasudevan et. al. mitigate microbursts caused by the Incast problem by changing TCP to use fine-grain timeouts [48], while ICTCP [49] seeks to prevent Incast by adjusting the receive window on the receiver side. As mentioned earlier, Partridge and Garrett [41] and Tang and Tai [45] did earlier work on tools for finding the token bucket parameters for a traffic stream that is related but different from our tools that measure burstiness.

4.6 Conclusion

This chapter is based on the premise that software pacing will become more important in a world where large bandwidth-intensive transfers compete with small latency-sensitive flows over commodity hardware and multi-gigabit links.

Although we have not stressed it, our experiments have shown that software pacing can be performed with modest processing overhead in a multi-core environment. For example, on our 8-core test machines, Linux TBF running at up to 10 Gbps incurred a CPU overhead of less than 12%. This is because a single core fielded the timer interrupts. Further, we showed that software pacing is susceptible to late timers, which can be exacerbated by system load. This manifests itself as a trade-off between reduced average bandwidth and increased burstiness. While larger bucket sizes can prevent a loss in average bandwidth for a single flow, this increases the queueing delay and latency experienced downstream by *many* other flows.

We have introduced a new measurement methodology to evaluate this trade-off and describe two new kernel-level tools, BandwidthTracker and BufferSim that can measure the burstiness of flows in software at very fine time scales, even in the presence

of coarse and inaccurate timers. Using these tools, we have evaluated the effectiveness of the Linux Token Bucket Filter (TBF) and found that, while TBF works very well at speeds of 1 Gbps or less, a number of issues appear at higher speeds. Upon further inspection, we discovered that kernel timers are negatively affected by system load and short expiration intervals. Timers were often late by 0.5 to 1μs, which caused significant loss in average bandwidth at low bucket sizes.

Our measurements revealed three simple areas for improvement. First, a lack of precision in division tables passed to the kernel from user-space caused considerable overshoot. Second, we found that TBF unnecessarily readjusted its timer, causing timers to be set for very short or negative durations, increasing contention, and ultimately wasting sending opportunities. Third, we found that TBF lost opportunities due to late timers and could compensate by occasionally sending packets slightly early.

We implemented these optimizations in the Linux kernel resulting in code that is more predictable (no overshoots), takes less software overhead (less timers set), and has a better trade-off between downstream buffer occupancy and average bandwidth (close to the target rate even at small bucket sizes). While much work remains to make a robust and accurate software pacing tool, we hope the tools and ideas in this chapter will provide a basis for future improvements.

Chapter 4, in part, is a reprint of the material being prepared for submission for publication. “How Effective is Software Pacing?”. Frank Uyeda, Amin Vahdat, George Varghese.

Chapter 5

Conclusions

In this dissertation, we have shown that the communication performance of large-scale distributed systems can be enhanced using software-based solutions. Toward this end, we have proposed three improvements: increasing communication efficiency through the use of Difference Digests, enabling fine-grain bandwidth monitoring to aid in tuning and debugging communication performance, and smoothing traffic at end hosts using software-based traffic pacing.

In Chapter 2, we have shown how Difference Digests can efficiently compute the set difference of data objects on different hosts using computation and communication proportional to the size of the set difference. Our contributions in this space are the application of Invertible Bloom Filters to the problem of computing set differences, and a new estimator that accurately estimates small set differences via a hierarchy of sampled IBF's. Through experimentation, we explored the parameter space for Difference Digests and provided guidance for the practical configuration of our algorithms.

Further, we implemented Difference Digests as part of our KeyDiff service that can be utilized by different applications to compute set differences over the network. Our system-level benchmarks exhibited the trade-offs between computation, storage, and communication and how they ultimately affect latency in various scenarios. We concluded that Difference Digests should be preferred over schemes that send the entire list of keys when the size of the difference is less than 20% and should be precomputed for best latency performance. In future work, we plan to use the KeyDiff service in real applications to improve overall user-perceived performance and hope that the simplicity,

elegance, and relevance of Difference Digests will inspire readers to more imaginative uses.

Next, in Chapter 3, we argued that today administrators have little information about when and how microbursts occur and need better visibility into the temporal dynamics and perpetrators of this behavior in order to tune their data-center environments. Our work provides the first steps toward realizing a cheap network-wide, bandwidth-usage database for identifying correlated burst patterns from across an administrative domain by introducing new, efficient summarization techniques. First, Exponential Bucketing offers accurate measurement of the average, maximum and standard deviation of bandwidths at arbitrary sampling resolutions with very low storage overhead. Additionally, bandwidth visualizations and more advanced statistics, such as quantiles, are provided by Dynamic Bucket Merge's time-series approximations. Ideally, these techniques will be supplemented by algorithms that also identify the flows responsible for bursts and techniques to join information across multiple links to detect offending applications and their timing.

While we have shown the application of `DBM` and `EXPB` to bandwidth measurements in end hosts, these algorithms could be easily ported to in-network monitoring devices or switches. Further, these algorithms can be generally applied to any time-series data, whether power, cooling, bandwidth, memory, CPU, or even financial markets, and will be particularly useful in environments where resource spikes must be detected at fine time scales but logging throughput and archival memory is constrained.

Finally, we explored software-based traffic pacing, a technique for rate-limiting and smoothing communication behavior at end hosts to avoid network congestion and microbursts. Chapter 4 is based on the premise that software pacing will become more important in a world where large bandwidth-intensive transfers compete with small latency-sensitive flows over commodity hardware and multi-Gigabit links.

To evaluate the effectiveness of software pacing, we have introduced two new kernel-level tools, BandwidthTracker and BufferSim that can measure the burstiness of flows in software at very fine time scales, even in the presence of coarse and inaccurate timers. We found that the Linux Token Bucket Filter works very well at speeds of 1 Gbps or less, but at higher speeds, and in the presence of high load, late timers create

a trade-off between reduced average bandwidth and increased burstiness. While larger bucket sizes can prevent a degradation of average bandwidth for a single flow, this can increase the queueing delay and latency experienced downstream by *many* other flows. Our observations led us to three simple improvements, which we implemented in the Linux kernel. The resulting implementation is more predictable, takes less software overhead, and has a better trade-off between downstream buffer occupancy and average bandwidth.

In summary, our contributions have shown software-based improvements to various facets of data-center communication with potential impacts for distributed computing in general. Even as our community pushes the boundaries of cloud computing and makes great strides with each new generation of hardware, we believe that software-based techniques for efficient communication will have continued importance in the decades to come.

Bibliography

[1] Cisco netflow. www.cisco.com/web/go/netflow.

[2] Facebook statistics. http://www.facebook.com/press/info.php?statistics.

[3] hrtimers - high-resolution timer subsystem. http://www.tglx.de/hrtimers.html.

[4] IEEE 802.1Qau - Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[5] iperf. http://iperf.sourceforge.net/.

[6] Linux Advanced Routing and Traffic Control. http://lartc.org.

[7] Performance Management for Latency-Intolerant Financial Trading Networks. *Financial Service Technology*, 9.

[8] pspacer. http://www.gridmpi.org/pspacer-2.1/.

[9] A simple network management protocol (snmp). http://www.ietf.org/rfc/rfc1157.txt.

[10] tcpdump. http://www.tcpdump.org/.

[11] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Communication, Control, and Computing, 2008*.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM*, 2010.

[13] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, 2000.

[14] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, 1970.

[15] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997*.

[16] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60:630–659, 2000.

[17] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. In *ICDE*, 2007.

[18] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks.

[19] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[20] D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: a high throughput transport protocol. In *SIGCOMM '87*.

[21] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *Proc. 31st Int. Conf. on Very Large Data Bases (VLDB)*, pages 25–36, 2005.

[22] M. Devera. http://luxik.cdi.cz/~devik/qos/htb/.

[23] D. Eppstein and M. Goodrich. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. on Knowledge and Data Engineering*, 23:297–306, 2011.

[24] A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.*, 4(2):209–223, 1996.

[25] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[26] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. of Computer and System Sciences*, 31(2):182 – 209, 1985.

[27] S. Gandhi, L. Foschini, and S. Suri. Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order. In *ICDE*, 2010.

[28] M. T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. *ArXiv e-prints*, January 2011. 1101.2245.

[29] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM 2009*.

[30] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive Spatial Partitioning for Multidimensional Data Streams. *Algorithmica*, 2006.

[31] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. STOC '98.

[32] S. Kalyanaraman, R. Jain, S. Fahmy, R. Goyal, and B. Vandalore. The ERICA switch algorithm for ABR traffic management in ATM networks. *IEEE/ACM Trans. Netw.*, 8(1):87–98, 2000.

[33] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC 2009*.

[34] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *SIGCOMM 2009*.

[35] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. Elephanttrap: A low cost device for identifying large flows. In *HOTI*, 2007.

[36] R. Martin. Wall Street's Quest To Process Data At The Speed Of Light. *Information Week*, April 23 2007.

[37] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *Information Theory, IEEE Transactions on*, 49(9):2213 – 2218, 2003.

[38] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[39] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[40] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1:117–236, 2005.

[41] C. Partridge. Manual page of TB program, 1994.

[42] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[43] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI 2011*.

[44] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. In *PFLDNet, 2005*.

[45] P. P. Tang and T.-Y. Tai. Network traffic characterization using token bucket model. In *INFOCOM '99*.

[46] F. Uyeda, L. Foschini, S. Suri, and G. Varghese. Efficiently measuring bandwidth at all time scales. In *NSDI 2011*.

[47] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. *SIGOPS Oper. Syst. Rev.*, 21(5), 1987.

[48] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM 2009*.

[49] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *Co-NEXT '10*.