SPACE-EFFICIENT DATA SKETCHING ALGORITHMS FOR NETWORK APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Nan Hua

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August, 2012

SPACE-EFFICIENT DATA SKETCHING ALGORITHMS FOR NETWORK APPLICATIONS

Approved by:

Prof. Jun (Jim) Xu, Adviser
School of Computer Science
Georgia Institute of Technology

Prof. Ellen W. Zegura
School of Computer Science
Georgia Institute of Technology

Prof. Nick Feamster
School of Computer Science
Georgia Institute of Technology

Prof. Justin Romberg
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Baochun Li
School of Electrical and Computer Engineering
University of Toronto

Date Approved: July 2, 2012

To

Jin and Xinya

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my advisor Jun (Jim) Xu, for his kind guidance and consistent support throughout my entire study. He is a great mentor, a great teacher and a great friend.

I would like to express my sincere gratitude to my other thesis committee members, Prof. Ellen Zegura, Prof. Nick Feamster, Prof. Justin Romberg and Prof. Baochun Li (Univ of Toronto), and the previous committee members Prof. Mostafa Ammar and Prof. Xiaoli Ma , for being great teachers and mentors, and for their interests in my graduate work and their valuable and insightful comments.

I would like to express my special gratitude to Prof. Justin Romberg, Prof. Baochun Li (Univ of Toronto), Prof. Bill Lin (UCSD), Prof. Yang (Richard) Yang (Yale) for their kind insightful guidance and hard work on our collaborated works.

I would like to thank my previous mentors in my previous internships, T.V.Lakshman and Haoyu Song (Bell Labs), Chuanxiong Guo (Microsoft Research Asia), Bill Lynch and Sailesh Kumar (Huawei US R&D Center), for providing opportunities to learn and broaden my horizons and support throughout these years.

I would like to say special thanks to my labmates and coauthors Haiquan (Chuck) Zhao, Ashwin Lall and Tongqing Qiu. I will always remember the delightful discussions with Chuck. I really appreciate the tremendous help and encouragement from Ashwin and Tongqing on my work and life.

I would like to thank all my student co-authors, especially Eric Norige (MSU) and Hao Wang (UCSD), for their tremendous helps and for our friendships.

My peers in the college, especially friends from the networking group deserve a special note of gratitude. For all this I thank Qi, Shuang, Cong, Valas, Murtaza, Yang, Samantha,

Yiyi, Srikanth and other folks from NTG.

Lastly but most importantly, I thank my parents, Jin Chen and Xinya Hua, for their love and sacrifice that made everything possible. I would not have come so far without their encouragement, understanding, and support, and to them I dedicate this dissertation.

TABLE OF CONTENTS

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

SUMMARY . . . xiii

I INTRODUCTION . . . 1

1.1 Highlights of this thesis . . . 3

1.1.1 Bloom Filter Alternative . . . 3

1.1.2 Exact Active Counter Array . . . 4

1.1.3 Error Estimating Coding . . . 5

1.2 Thesis Organization . . . 6

II RANK-INDEXED HASHING: COMPACT CONSTRUCTION AND ENHANCEMENT OF BLOOM FILTER . . . 8

2.1 Problem Overview . . . 8

2.2 Background and Related Works . . . 11

2.2.1 Bloom Filters, Counting Bloom Filter and Compressed Bloom Filter . . . 11

2.2.2 Fingerprint Hash Table Construction . . . 12

2.3 Rank-Indexed Hashing . . . 14

2.3.1 Basic Idea . . . 14

2.3.2 Packed Bucket Organization . . . 18

2.3.3 Quasi-Dynamic Bucket Sizing . . . 19

2.4 Tail Bound Analysis . . . 22

2.5 Evaluation . . . 26

2.6 Combination with *d*-left Hashing . . . 29

2.6.1 Analysis of Memory Utilization of *d*-left Bloom Filter . . . 30

2.6.2 Rank-Indexed *d*-left Bloom Filter . . . 34

2.6.3 Evaluation and Discussion of Rank-Indexed d-left Bloom Filter . 36
2.7 Conclusion 38
III BRICK: RANK-INDEXING TECHNIQUE FOR EXACT ACTIVE STATISTICS COUNTER ARRAY ARCHITECTURE 39
3.1 Problem Overview 39
3.1.1 Our approach and contributions 40
3.2 Background and Related works 44
3.3 Design of BRICK 45
3.3.1 Overview 46
3.3.2 Rank Indexing 49
3.3.3 Handling Increments 50
3.3.4 Handling Overflows 52
3.4 Analysis 54
3.4.1 Analytical Guarantees 54
3.4.2 the Main Tail Bound 56
3.4.3 Proofs of propositions 1–3 59
3.5 Information theory bound 61
3.5.1 The worst-case average binary length of counters 61
3.5.2 Bound_1: Considering the unavoidable indexing costs 62
3.5.3 Bound_2: Lower bound when optimal coding is used 63
3.6 Performance Evaluations 65
3.6.1 Numerical Results of Analytical Bounds 65
3.6.2 Results for real Internet traces 69
3.7 Discussions 71
3.7.1 Implementation issues 71
3.8 Conclusion 72
IV DESIGN AND ANALYSIS OF ERROR ESTIMATING CODING: NEW SKETCHES, ESTIMATORS AND ANALYSIS FRAMEWORK 74
4.1 Problem Overview 74

4.2 Background, Preliminaries and Related Works . . . 77
4.2.1 Error Estimating Codes (EEC) . . . 78
4.2.2 Randomized Approximation and Communication Complexity . . 80
4.2.3 Fisher Information and Cramer-Rao Bound . . . 81
4.2.4 Data Streaming and Communication Complexity . . . 83
4.2.5 Definitions and notes of Notations . . . 84
4.3 Lower Bounds . . . 84
4.3.1 Why Randomization Is Needed . . . 84
4.3.2 Why Approximation is Needed . . . 86
4.3.3 Randomized Approximation . . . 87
4.4 Tug-of-War Sketch for Error Estimating Coding . . . 87
4.4.1 The sketch . . . 87
4.4.2 Analysis . . . 89
4.4.3 Cost and Overhead for EEC applications . . . 90
4.5 Enhanced Tug-of-war Sketch (EToW): Scheme and Analysis . . . 91
4.5.1 Analysis of the effect of sampling . . . 94
4.5.2 Analysis of truncation and sampling together . . . 95
4.5.3 Impact of bit errors on counters and protection . . . 98
4.6 Fisher Information Analysis of EEC . . . 100
4.6.1 Fisher Information Contained in each EEC bit . . . 102
4.6.2 Combining the contributions of the different levels of bits . . . 105
4.6.3 MLE estimator for EEC scheme . . . 106
4.7 Design and Fisher Information Analysis of Generalized EEC (gEEC) . . 106
4.7.1 The gEEC Encoding . . . 107
4.7.2 Fisher information analysis . . . 109
4.7.3 Our MLE Estimators . . . 114
4.7.4 Numerical Results of the Fisher Information contained in the gEEC Family . . . 114
4.8 Evaluation . . . 116

4.8.1 Experimental Results 116
4.8.2 Implementation Cost of Estimators and Selection of Parameters . 120
4.9 Conclusions 122
V CONCLUSION AND FUTURE WORKS 130
REFERENCES 134
VITA 139

LIST OF TABLES

1 Table entries give the fraction of buckets with actual load greater than some threshold for a bucket with $L = 64$ hash chain locations and $\lambda = 0.875$. Asymptotic tail bounds are provided for three example thresholds. 20

2 Representative results for rank-indexed hashing under various parameter settings ($P_o = 10^{-10}$, $n = 10^5$). 27

3 Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the standard Bloom filter function. 29

4 Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the counting Bloom filter (CBF) function. 29

5 Comparison of transferring cost (in bits) per element against Compressed Bloom Filter and d-left Bloom Filter 30

6 Representative results for rank-indexed d-left hashing Bloom Filter 37

7 Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the standard Bloom filter function. 37

8 Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the counting Bloom filter (CBF) function. 37

9 Bounds for schemes without using any coding technique 63

10 Information-theoretic lower bound. 64

11 An Example of Sub-counter array sizing and per-counter storage for $k = 64$ and $P_f = 10^{-10}$. 67

12 Percentage of full-size buckets. 70

13 Definition of symbols for Error Estimating Problem 85

14 The total information gain $\frac{SJ}{kC}$ ($C = \frac{1}{24}\pi^2$) 116

LIST OF FIGURES

1 Hash table with chaining. . . . 13

2 Buckets and hash chain locations. . . . 15

3 Rank-indexed hashing. . . . 17

4 Quasi-dynamic bucket sizing via bucket extensions. . . . 21

5 Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the standard Bloom filter function. . . . 31

6 Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the counting Bloom filter (CBF) function. . . 32

7 memory savings through adjusting the size of bucket in d-left Bloom Filter, when targeted false positive rate is 0.4% 34

8 An illustration of d-left Bloom Filter 35

9 BRICK wall (conceptual baseline scheme) 42

10 Randomly bundling counters into buckets. . . . 47

11 (a) Within a bucket, segmentation of variable-width counters into sub-counter arrays. (b) Compact representation of variable-width counters. (c) Updated data structure after incrementing C_2. . . . 48

12 Impact of increasing bucket size k. Extra bits σ in the range of $[5.03, 6.05]$. 68

13 Impact of increasing number of flows N. Extra bits σ in the range of $[5.65, 5.78]$. . . . 69

14 Impact of increasing $\log_2 \frac{M}{N}$. Extra bits σ in the range of $[5.62, 5.69]$. . . . 69

15 Impact of decreasing failure probability P_f. Extra bits σ in the range of $[5.65, 5.70]$. . . . 69

16 Overview of Enhanced Tug-of-War (EToW) Sketch (Algorithm 3) 92

17 Comparison of the variance of the sampled tug-of-war sketches and the original one, with $c = 16$. . . . 95

18 The relative Rooted Mean Squared Error of the enhanced tug-of-war sketch (fully protected) with different sampling and truncation parameters, when $c = 16, l = \{512, 1024\}, k = \{4, 5, 6\}$. . . . 99

19 Comparisons of different constructions of parity-checking bits. When $k = 5$, we use sampling length $l = 512$; when $k = 6$, we use $l = 1024$ 101

20 Fisher Information (of $\log \theta$) of each EEC bit in function of l and θ 102

21 the empirical performance of EEC's original estimators and the associated Cramer-Rao Bound (all levels, each level, and the envelope of only one level/two levels). The EEC scheme is composed by nine levels and 32 bits each level. 124

22 gEEC's Fisher information: Relationship to l and k in the case with immunity assumption 125

23 gEEC's Fisher information: Relationship to l and k in the cases with and without immunity (denoted as *w.t.i.* and *w.o.i.* respectively in the legend) . . 125

24 Experimental Results of EToW sketch 126

25 Empirical Results of Estimators: Compare the performance of original EEC (with original estimator), two gEEC schemes (with much smaller size, 80 and 96 bits respectively), and one EToW scheme (with 96 bits), in the case without "immunity". 127

26 Empirical Results of Estimators: Compare the performance of two types of gEEC estimators (with or without Jeffrey's prior) on three schemes including the original EEC scheme, in the case without "immunity". 128

27 Empirical Results of Estimators: Compare the performance of four schemes (original EEC, gEEC(56,512,5),gEEC(48,768,6) and EToW), with almost the same size and using gEEC's estimator with Jeffrey's prior, in the case without "immunity". 129

SUMMARY

Sketching techniques are widely adopted in network applications. Sketching algorithms "encode" data into succinct data structures that can later be accessed and "decoded" for various purposes, such as network measurement, accounting, anomaly detection and etc. Bloom filters and counter braids are two well-known representatives in this category. Those sketching algorithms usually need to strike a tradeoff between performance (how much information can be revealed and how fast) and cost (storage, transmission and computation). This dissertation is dedicated to the research and development of several sketching techniques including improved forms of stateful Bloom Filters, Statistical Counter Arrays and Error Estimating Codes.

Bloom filter is a space-efficient randomized data structure for approximately representing a set in order to support membership queries. Bloom filter and its variants have found widespread use in many networking applications, where it is important to minimize the cost of storing and communicating network data. In this thesis, we propose a family of Bloom Filter variants augmented by rank-indexing method. We will show such augmentation can bring a significant reduction of space and also the number of memory accesses, especially when deletions of set elements from the Bloom Filter need to be supported.

Exact active counter array is another important building block in many sketching algorithms, where storage cost of the array is of paramount concern. Previous approaches reduce the storage costs while either losing accuracy or supporting only passive measurements. In this thesis, we propose an exact statistics counter array architecture that can support active measurements (real-time read and write). It also leverages the aforementioned rank-indexing method and exploits statistical multiplexing to minimize the storage

costs of the counter array.

Error estimating coding (EEC) has recently been established as an important tool to estimate bit error rates in the transmission of packets over wireless links. In essence, the EEC problem is also a sketching problem, since the EEC codes can be viewed as a sketch of the packet sent, which is decoded by the receiver to estimate bit error rate. In this thesis, we will first investigate the asymptotic bound of error estimating coding by viewing the problem from two-party computation perspective and then investigate its coding/decoding efficiency using Fisher information analysis. Further, we develop several sketching techniques including Enhanced tug-of-war(EToW) sketch and the generalized EEC (gEEC)sketch family which can achieve around 70% reduction of sketch size with similar estimation accuracies.

For all solutions proposed above, we will use theoretical tools such as information theory and communication complexity to investigate how far our proposed solutions are away from the theoretical optimal. We will show that the proposed techniques are asymptotically or empirically very close to the theoretical bounds.

CHAPTER I

INTRODUCTION

In this dissertation research, we will focus on analyzing and improving several "sketching" algorithms which are important in a wide variety of network applications. Those sketching algorithms "encode" the information of packets into some succinct data structures which will be later accessed and "decoded" for various purposes, such as network measurement, accounting and anomaly detection. Bloom Filter, statistics counter array and error estimating codes are three such sketch data structures, which will be the focus of this dissertation research. Bloom Filter[5] is a data structure which approximately encodes the information of a set for set membership queries. Applications which employs the Bloom Filter can query the Bloom Filter to "decode" whether an arbitrary item belongs to the Bloom Filter. Statistics counter array (such as counter braids [46]) is a data structure that encodes the values of an array of statistics counters approximately or accurately. Applications which employ this data structure need to "decode" the counter value(s) either online or offline. Error estimating code [13] is a data structure which partially encodes the information of a packet, in such a way that if certain number of bits in the packet are flipped later on, this number can be estimated from the code. In summary, those sketching algorithms all succinctly encodes certain information about packets for later accesses.

Those sketching algorithms play an fundamentally important role in overcoming major resource bottlenecks in networking applications. The most typical bottleneck in high-speed router applications is the limited amount of expensive on-chip memory (SRAM), the limited bandwidth and high latency between on-chip processors and off-chip memory, and the limited off-chip memory size. In web caching and network measurement [9, 22, 10], Bloom Filters[5] are therefore employed to filter out invalid lookups to the slower external

or remote memories or disks, which will greatly save the bandwidth to the slower storage devices. As for the statistics counter array problem, since it is too costly to naïvely implement the counter array on the scale of millions of flows, several different algorithms, such as counter braids [46], approximate counting[17] , have been proposed to make more efficient utilizations of the precious on-chip resources.The last problem we will visit in this thesis is the error estimating code problem[13], which was proposed to enable the receiver to directly infer bit error rate from the received packet and codes, overcomes the limitation that the MAC layer can only infer the bit error rate from either indirect sources such as packet loss ratio or S/N ratio or from some special hardware such as the SoftPHY in [64, 36].

Due to such resource limitations, the implementation of those sketching algorithms need to strike a tradeoff between performance (how much information can be revealed and how fast) and cost (storage, transmission and computation). Although the three sketching problems listed above, Bloom Filter, counter array and error estimating code, look different from the perspective of either design or application area, and the limitations/bottlenecks faced by those three applications are also not identical, they share similar characteristics from an algorithmic perspective: smaller sketching size is important (with the same or better functionalities) for saving either the precious on-chip resources or the transmission overhead.

Although the sketching algorithms mentioned above are motivated and proposed already with better space efficiency in mind, we found that they are far from the optimal, especially when some additional functionalities need to be supported. The theme of this dissertation research is not to propose new sketching problems, but to explore the space-saving and performance-improving spaces for those established important sketching problems from a practical algorithmic perspective. In short, we aim to design space-efficient solutions for the three problems listed above.

In general, for all three problems listed above, we will follow the same methodology.

First, we will thoroughly analyze the existing solution(s) and identify some promising directions. Second, we will design new solution(s) often with intuitions and techniques brought in from some other areas. We will see that although the underlying problems are very different from one another, they share some key ideas and techniques such as "statistical multiplexing" and rank-indexing.

Third, we will rigourously analyze the average or statistically worst-cases of the scheme and will optimize and evaluate our proposed solutions based on these analysis results. Finally, we will explore the question of how far the proposed solutions are from the optimal in theory, which might shed light on further innovations.

In the next three subsections, we will highlight our storylines on the three aforementioned sketching problems followed by organization of the thesis.

1.1 Highlights of this thesis

1.1.1 Bloom Filter Alternative

We first apply the methodology presented above on the Bloom filter problem. Bloom Filter is a space-efficient randomized data structure for approximating a set in order to support membership queries. Bloom filter and its variants have found widespread use in many networking applications [9]. The most classical and typical application scenario of Bloom Filter serves as pre-lookup filter before a lookup into a large table in slower memory/disk. Bloom filter can filter out most of the invalid requests (requests for items that are not in the table) and hence be able to greatly reduce the traffic into slower memory/disk. Bloom Filter is also widely employed in many newly proposed designs, including web caching, packet/resource routing, P2P collaborating, network measurement, deep packet inspection, etc. Whenever and wherever a succinct data sketch is needed and approximation is allowed, usually Bloom Filter will be a good candidate, while how much improvement in performance can be achieved depends on the particular application scenario. In most applications scenarios of Bloom Filter, it is important to reduce space cost.

The original design of Bloom Filter is just a binary array with k hash functions and supports insertion only. Due to its wide applicabilities, many variants, such as counting Bloom filter (CBF) [22], Spectral Bloom Filter [14], the Approximate Concurrent State Machine [6] (also called a stateful Bloom filter), the Bloomier filter [11], etc, have been proposed to support versatile additional functionalities and can be optimized for different application scenarios.

Due to the fundamental importance of Bloom Filter, we start this dissertation work by re-visiting the Bloom Filter design and find that the Bloom Filter problem should have the potential to be further improved from the memory-saving perspective. We start with an alternative approach to Bloom filter design, namely the fingerprint hash-table, and augment it with a special bit-operation trick called rank-indexed hashing. We will show this family of constructions result in a significant reduction in space and less memory accesses and better memory access locality, especially when deletions need to be supported. We will also employ some large-deviation techniques to perform a tail-bound analysis of the proposed scheme.

1.1.2 Exact Active Counter Array

The next problem we will visit is the design of exact active counter array. Statistical counter array is another important building block in many sketching applications, where is also important to save storage cost. Large-scale array of counters (millions or more) are needed in schemes ranging from the basic packet counting features of core network devices to some advanced network data streaming algorithms for network measurement and troubleshooting. Previous approaches can reduce the storage cost in cost of either losing accuracy or only supporting passive measurements[56, 53, 55, 70, 49, 17, 58].

In Chapter 3, we will propose an exact and active solution. Similar to the Bloom filter built through rank-indexed hashing, our proposed solution also leverages the rank-indexing method and exploits statistical multiplexing to save the storage costs of counter values. A

tail-bound analysis of the worst case scenarios is provided. Different from the Bloom Filter case, the information theory limit of the counter array problem is non-trivial and hence we will research this limit from a few different angles.

1.1.3 Error Estimating Coding

The third problem we will visit in this dissertation is the design and analysis of error estimating coding. Error estimating coding (EEC) [13] has recently been established as an important tool to estimate bit error rates in the transmission of packets over wireless links. The concept of EEC breaks the long-held design philosophy that only wants to deal with fully correct data, through the correction capability provided by error correcting code (ECC). In contrast of ECC, EEC aims to use much smaller overhead to only estimate the number of errors while not correct them. It was shown in [13] that, if the BER in packets can be accurately estimated, important operations in wireless networks such as packet re-scheduling, routing, and carrier selection can all be performed with greater efficiency.

In essence, the EEC codeword is a sketch of the packet. The receiver will decode the binary error rate from the packet received by "comparing" the received data part and sketch part. Following the same methodology as the previous problems we have visited, it's natural to ask how good is the existing solution proposed in [13] and whether possible to improve it. In Chapter 4, we visit the EEC problem from a few different angles. Firstly, we cast this BER estimation problem into the rich theoretical framework of two-party computation. This perspective will bring us a proof of the asymptotic optimality of the original scheme and also intuitions in designing new sketch. We found that a classical solution to the two-party computation of hamming distance, tug-of-war sketch, although not directly applicable, can be leveraged to build a new sketching scheme, which we called Enhanced tug-of-war (EToW) sketch. EToW sketch can deliver similar performance with around 60% reduction of size, along with some additional benefits such as simpler estimator and numerically predictable performance.

Secondly, we follow up with a deeper and more important question whether EEC has achieved the desired optimal space-accuracy tradeoff. We leveraged the Fisher information analysis tool to analyze the potential of the original EEC design and find that actually the variance achieved by the original EEC's estimator is much larger than the Cramer-Rao lower bound, which means there's a lot of inefficiency inside the original EEC's estimator. We hence proposed a new estimator which is close to Cramer-Rao bound empirically. Moreover, we further find that the EToW sketch newly proposed by us and the original EEC sketch can subsumed into a generalized family of sketches, which we call generalized-EEC (gEEC) family. We developed an analysis framework and estimator for gEEC, which shed light on deeper understanding of the problem. Through the unified framework of gEEC, we found that some parameterizations of gEEC (similar to EToW, but not needing the extra sketch error detection bits) can further improve the estimation efficiency by another 25-35 % under certain circumstances. Moreover, gEEC can be flexibly configured for different scenarios than its two "degenerate" cases (EEC and EToW).

1.2 Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, we will present our rank-index hashing construction of Bloom Filter. We will first overview the problem in Section 2.1. Section 2.2 summarizes background material on Bloom filters and fingerprint hash table constructions. Section 2.3 describes our rank-indexed hashing method, which will be also the foundation of the BRICK scheme to be presented in Section 3.3. Section 2.4 establishes tail bound probabilities that allow us to bound and optimize the storage cost. Section 2.5 evaluates our scheme by presenting numerical results under various parameter settings, including results for both standard and counting Bloom filters. Section 2.6 combines the rank-indexing with the d-left hashing to obtain more savings in memory cost.

Chapter 3 presents our rank-indexing-based statistics counter architecture. We firstly overviews the problem in Section 3.1 . In Section 3.3, we describe the design of our scheme

in detail. In Section 3.4, we establish the tail probabilities that allow us to bound and optimize the SRAM requirement. In Section 3.5, we derive several lower bounds on memory usage for counters to help us understand how far we are from the optimal. In Section 3.6, we evaluate our scheme by presenting numerical results on memory costs and tail probabilities under various parameter settings, including those extracted from real-world traffic traces.

Chapter 4 presents our approaches and results on error estimating coding. The EEC problem and our approaches are firstly overviewed in Section 4.1.Problem statement, notations, some background presentation and the related work most pertinent to this dissertation work are presented in Section 4.2. In section 4.3 we analyze the asymptotic lower bounds of error estimating codes, in terms of the number of overhead bits needed. Section 4.4 describes the tug-of-war sketch, and gives a simple analysis to show that it can accurately computes BER if the sketch is not corrupted by errors. In Section 4.5, we propose our enhanced tug-of-war sketch that removes the assumption of integrity of the sketch and substantially improves its performance. In Section 4.6, we move on to use the Fisher information tool to analyze the original EEC algorithm. In Section 4.7, we propose the generalized EEC scheme and provide the corresponding analysis, estimator design and numerical results. We evaluate the performance experimentally in Section 4.8.

We conclude the thesis and discuss possible directions of future work in Chapter 5.

CHAPTER II

RANK-INDEXED HASHING: COMPACT CONSTRUCTION AND ENHANCEMENT OF BLOOM FILTER

2.1 Problem Overview

A Bloom filter [5] is a space-efficient randomized data structure for approximating a set in order to support membership queries. Although a Bloom filter may yield false positives, saying an element is in the set when it is not, it provides a very compact representation that can be configured to achieve sufficient accuracies for many applications. For a false positive probability of ϵ, an optimal configuration only requires $1.44(\log 1/\epsilon)$ bits per element, independent of the number of elements in the set. For example, to achieve a false positive probability of $\epsilon = 1\%$, only 10 bits of storage per element is required.

In recent years, there has been a huge surge in the popularity of Bloom filters and variants, especially for network applications [9]. One variant is the counting Bloom filter (CBF) [22], which allows the set to change dynamically via insertions and deletions of elements. Other generalizations of Bloom filters include Spectral Bloom Filter [14], which can encode approximate counts, the Approximate Concurrent State Machine [6] (also called a stateful Bloom filter), which can encode state information, and the Bloomier filter [11], which can encode arbitrary functions by allowing one to associate values with a subset of the domain elements. In general, many Bloom filter variants that permit the association of values to elements mainly differ in the way how they encode and interpret the values associated.

Although a standard Bloom filter construction is very space-efficient for simple membership queries, it is actually rather inefficient when generalized to support deletions or the encoding of information. In particular, in the standard Bloom filter construction, an array

of m bits is used to represent a set S of n elements, where m is chosen to be sufficiently large to ensure a small false positive probability. For example, for a false positive probability of $\epsilon = 1\%$, m is chosen to be 10 times n, resulting in an amortized storage cost of 10 bits per element. When this standard construction is generalized to encode additional information, an array of m locations is used instead of bits. For example, in the counting Bloom filter application where a counter is associated with each location to support insertions and deletions, four counter bits are often used to provide a sufficient level of accuracy [22]. However, this blows up the storage requirement by a factor of four over a standard Bloom filter.

Alternatively, it is well-known that a hash table construction with fingerprints can be used to provide the same functionality as a Bloom filter [9]. In particular, if the set S is static and a perfect hash function can be constructed for a hash table with n locations, then storing a fingerprint with only $\lceil \log 1/\epsilon \rceil$ bits for each element at the corresponding location would suffice to achieve a false positive probability of ϵ. Moreover, for Bloom filter generalizations that support values associations, the encoding of the additional information only needs to be stored once at the corresponding hash table location rather than requiring the encoding of information across multiple locations as required in a standard Bloom filter construction, resulting in substantial savings in space. However, unfortunately, perfect hashing is very difficult to construct and does not support dynamically changing sets.

In this thesis, we propose a new fingerprint hash table construction called *Rank-Indexed Hashing* that provides a compact replacement for Bloom filters, counting Bloom filters, and other Bloom filter variants. Conceptually, our starting point is a conventional chaining-based hash table scheme. However, our proposed solution avoids the costly overhead of pointer storage by employing an efficient indexing scheme called *rank-indexing*. Actually rank-indexing is not a brand-new technique and has been employed by [18] and [59] to construct compact data structures. The name "rank-indexing" here is from [4], which summarizes this kind of operation as rank operation. Using rank-indexed hashing construction,

we show that it is possible to outperform a standard Bloom filter construction in storage cost for false positive probabilities at or below just 0.1%, which is significant since a standard Bloom filter construction is widely regarded as a very space-efficient data structure for approximate membership query problems, and it is often desirable to have a false positive probability smaller than 0.1% in many applications.

For the counting Bloom filter application, the rank-indexed hashing construction is able to outperform a standard counting Bloom filter construction in storage cost by a factor of three for a false positive probability of just 1%, and it is able to outperform a recently proposed fingerprint hash table construction called d-left hashing [7] in storage cost by 27% at the same false positive probability. Similar storage cost benefits are expected for other Bloom filter variants. Especially for network applications, smaller storage requirement is a central design metric because Bloom filters are often implemented using relatively scarce and expensive (on-chip) SRAM. Although SRAM capacity continues to increase, the rate of traffic growth continues to outpace transistor density, leading to an ever increasing need to reduce storage requirements.

Since rank-indexing is a technique almost orthodox to d-left hashing, we could combine those two techniques together to get even more memory savings. However, the memory accesses are slightly increased and the performance guarantee is also not as strict as that of the pure rank indexed hashing.

Rank-indexed hashing also has advantages against the well-known Compressed Bloom Filter[47]. When used for transferring purpose, Bloom Filters constructed by rank-indexed hashing could be transferred even more compactly. If using Compressed Bloom Filter, it would need a very large original data array to achieve the same size for transferring.

2.2 Background and Related Works

2.2.1 Bloom Filters, Counting Bloom Filter and Compressed Bloom Filter

A Bloom filter represents a set S of n elements from a universe U using an array of m bits, denoted by $\phi[1], \ldots, \phi[m]$. Initially, all positions $\phi[i]$ are set to 0. A Bloom filter uses a group of k independent hash functions $h_1, \ldots, h_k$ with range $[m] = \{1, \ldots, m\}$. Each hash function independently maps each element in the universe to a random number chosen uniformly over the range. For each element $x \in S$, the bits $\phi[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. To check if an item x is in S, we check whether all $h_i(x)$ are set to 1. If not, then by construction, x is not a member of S. If all $h_i(x)$ are set to 1, then x is assumed to be in S, which may yield a *false positive*.

The false positive probability for an element not in the set can be computed as

$$\epsilon = \left(1 - (1 - 1/m)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

The storage requirement to satisfy a given false positive probability ϵ is minimized when

$$m = \frac{1}{\ln 2} kn \approx 1.44kn.$$

For example, when $m/n = 10$ and $k = 7$, the false positive probability is about $2^{-7} \approx 0.008$.

A Bloom filter allows for easy insertion, but not deletion. Deleting an element from a Bloom filter cannot be done simply by reverting the corresponding ones back to zeros since each bit may correspond to multiple elements. Deletion can be handled by using a counting Bloom filter (CBF) [22], which uses an array of m counters instead of bits. Counters are incremented on an insertion and decremented on a deletion. The counters are used to track approximately the number of elements currently hashed to the corresponding locations. To avoid overflow, counters must be chosen to be large enough. For most applications, 4 bits per counter have been shown to suffice [22]. However, the obvious disadvantage of counting Bloom filters is that they appear to be quite wasteful of space. Using counters of 4 bits blows up the storage requirement by a factor of four over a standard Bloom filter,

even though most counters will be zero. For example, with $k = 7$, $4m/n = 40$ bits per element are needed to achieve a false positive probability of about $2^{-7} \approx 0.008$.

Compressed Bloom Filter [47] is in fact a special application of standard Bloom Filter, rather than a new data structure. It is discovered in [47] that, when the space of standard Bloom Filter is minimized to $1.44kn$, its compressed size, i.e. its information entropy, is also maximized at the same time. In another word, if the space of standard Bloom Filter is not optimized to the minimum, i.e. greater than $1.44kn$, its size after compression would be smaller than $1.44kn$. The larger uncompressed size, the smaller compressed size. When the original uncompressed space is infinitely large, the compressed size would reach the limit kn. Hence, we could sacrifice the original space for smaller compressed size. This property is favored for some online applications such as P2P sharing where the transferring cost is much more important.

2.2.2 Fingerprint Hash Table Construction

An alternative construction of Bloom filters and counting Bloom filters is to use a hash table with *fingerprints*. It is well known that if the set S is static, then one can achieve essentially optimal performance by using a perfect hash function and fingerprints [9]. That is, we can find a perfect hash function

$$P : U \rightarrow [n]$$

that maps each element $x \in S$ to a unique location in an array of size n, where $[n]$ denotes the range $\{1, \ldots, n\}$. Then, we simply need to store at each location a fingerprint with $\lceil \log 1/\epsilon \rceil$ bits that is computed according to some hash function F. A query on x requires computing $P(x)$ and $F(x)$, and checking whether the fingerprint stored at $P(x)$ matches $F(x)$. When $x \in S$, a correct response is given. But when $x \notin S$, a false positive occurs with probability at most ϵ. This perfect hashing approach achieves the optimal space requirement of $\lceil \log 1/\epsilon \rceil$ bits per element. However, the problem with this approach is that it does not allow the set S to change dynamically, and perfect hash functions are generally

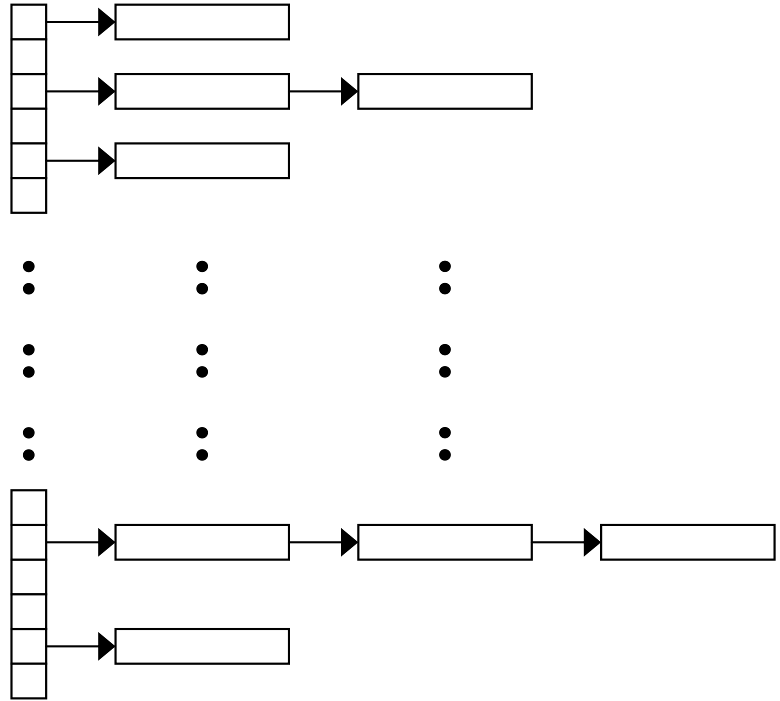


Figure 1: Hash table with chaining.

very difficult to compute for most applications.

An alternative to perfect hashing is to use an *imperfect* hash function. Suppose we use a single (imperfect) hash function $H : U \rightarrow [B]\times[R]$ to hash the elements in S to a hash table with B buckets. For each element $x \in S$, $H(x)$ returns two parts. The first part in the range $[B]$ corresponds to the bucket index in which the element should be placed. The second part in the range of $[R]$, referred to as the *remainder*, corresponds to a *compressed fingerprint* that gets stored in the corresponding bucket. Using a single hash function, it is possible (and likely) that two different elements x and y are mapped to the same bucket, resulting in a *collision*. One way to resolve collisions is to allocate a fixed number of *cells* per bucket so that the maximum load per bucket is no more than this fixed number with high probability. However, the distribution of load using a single hash function can fluctuate dramatically across buckets, leading to a lot of wasted space.

Another way to resolve collisions is to maintain a dynamically allocated *linked list* of fingerprints that have been hashed to the same bucket, as shown in Figure 1. However, this

conventional *chained hashing* approach is also rather inefficient in that the extra pointer storage (required for each fingerprint) is very expensive. For $n = 1$ million elements and a desired false positive probability of $\epsilon = 2^{-7}$, 7 bits per element would suffice assuming perfect hashing, but another $\lceil \log n \rceil = 20$ bits would be required on average per element to implement pointers, or another 40 bits per element to implement doubly linked list pointers to support deletion, increasing the storage requirement by nearly seven-fold.

A third way that has been recently proposed [7] is to use a balanced allocation approach due to Vöcking [62, 63] called d-left hashing. By splitting a hash table into multiple equally-sized subtables, and placing elements in the least-loaded subtable, d-left hash tables can be dimensioned statically so that the average load per bucket is close to the maximum load. A good configuration suggested in [7] is to use 4 subtables with a fixed allocation corresponding to an expected maximum load of 8 fingerprints per bucket, with an expected average load of 6 fingerprints per bucket. To check if x is in S, d-left hashing requires checking x against all fingerprints stored in the corresponding buckets across all the subtables, requiring the retrieval of $4 \cdot 8 = 32$ fingerprints, with matching on average against $4 \cdot 6 = 24$ fingerprints expected. To achieve a desired false positive probability of $\epsilon = 2^{-7}$, each cell must store a fingerprint with $\lceil \log 24/\epsilon \rceil = \lceil \log 24/2^{-7} \rceil = 12$ bits, adding 5 more bits per element to the "ideal" case of $\lceil \log 1/\epsilon \rceil = 7$ bits, which is significant. Further accounting for the expected fraction of unused cells corresponding to the ratio of expected maximum load over average load, $(8/6) \cdot 12 = 16$ bits per element would be required, increasing over the ideal storage requirement of 7 bits by over two-fold.

2.3 Rank-Indexed Hashing

2.3.1 Basic Idea

In this section, we describe our proposed *rank-indexed hashing* approach. Conceptually, our starting point is a conventional chaining-based hash table scheme. However, we employ a *two-level* indexing scheme by using a single hash function $H : U \rightarrow [B] \times [L] \times [R]$ that

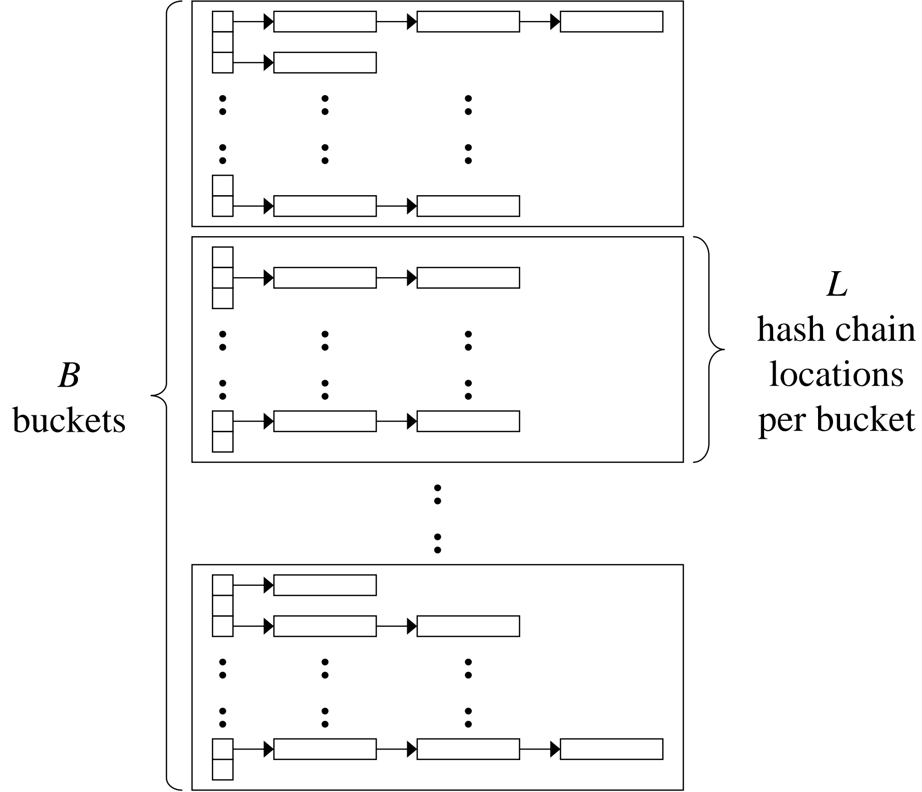


Figure 2: Buckets and hash chain locations.

hashes each element in S into three parts; i.e., for each element $x \in S$, $H(x) = (b, \ell, r)$. As before, the first part b in the range of $[B]$ corresponds to the bucket index in which the element should be placed. The second part ℓ in the range of $[L]$ corresponds to a *hash chain* location within a bucket. This is conceptually depicted in Figure 2. The third part r in the range of $[R]$ corresponds to the (compressed) fingerprint that gets stored in the corresponding hash chain location. In general, the number of buckets B times the number of hash chain locations per bucket L can be different than the number of elements n. We use

$$\lambda = \frac{n}{BL}$$

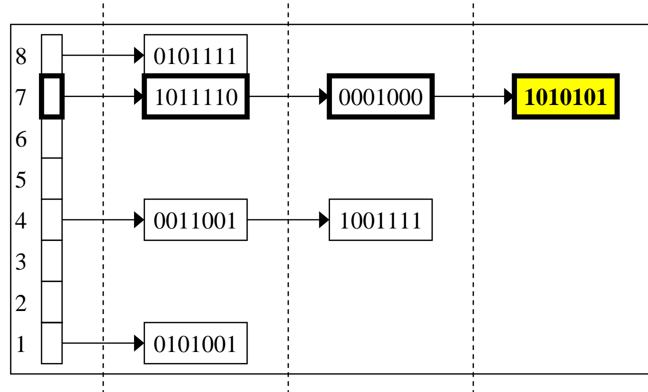
to denote the expected average load per hash chain location. If $\lambda = 1$, then a fingerprint with $\lceil \log 1/\epsilon \rceil$ bits suffices to achieve a false positive probability of ϵ. In general, $\lceil \log \lambda/\epsilon \rceil$ bits are needed to achieve a false positive probability of ϵ.

Given λ, the expected average load per bucket is simply $\lambda \cdot L$. Intuitively, each bucket is dimensioned to store a fixed number of $Z > (\lambda \cdot L)$ fingerprints such that the actual load per bucket is less than or equal to Z in a large fraction of buckets. We defer to Section 2.3.3 to discuss the handling of cases when the actual load exceeds Z fingerprints. For the moment, we will assume Z is chosen to ensure that actual load is less than or equal to Z with high probability.

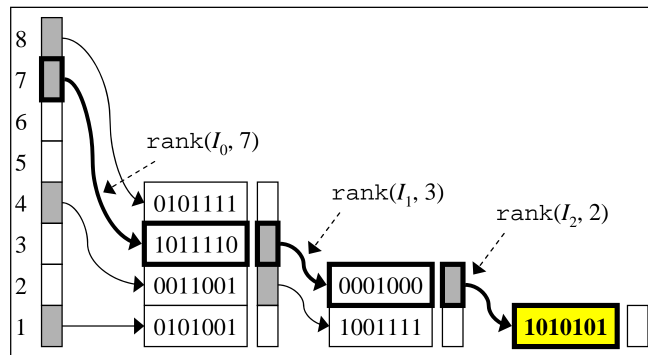
A key innovation in our proposed approach is an indexing method that allows us to efficiently realize *dynamic chaining* at each hash chain location *without* the costly overhead of pointer storage. We call this method *rank-indexing*, and this method is illustrated in Figure 3. Consider a bucket with $L = 8$ hash chain locations, as shown in Figure 3(a). In this example, suppose the longest chain has three fingerprints. We can conceptually partition the fingerprints into three levels, in accordance to the depths of the fingerprints in the corresponding chains. We then *pack* the fingerprints at each level together in a corresponding contiguous subarray of fingerprints.

To locate fingerprints in a bucket, we maintain an index bitmap I. Conceptually, I is divided into multiple parts $I_0, I_1, \ldots, I_d$, one part for each level of subarray. Suppose we want to query for the fingerprint "1010101" at the hash chain location $\ell = 7$. We first check $I_0[7]$ to determine if there are fingerprints stored at $\ell = 7$. If $I_0[7]$ is set to 1, as shown in a shaded box in Figure 3(b), then it means there is a non-empty chain at $\ell = 7$. If there are no fingerprints at location ℓ, then the corresponding $I_0[\ell]$ would be set to 0, which is shown as a clear box.

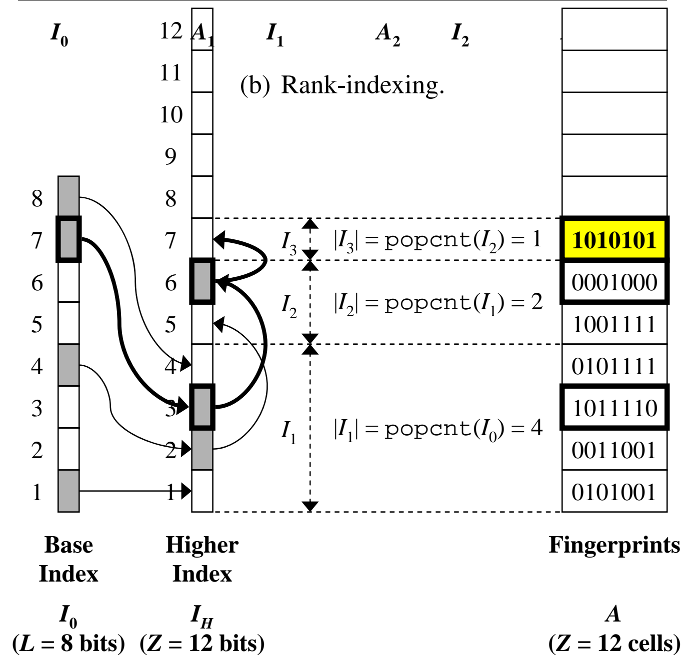
In the example shown in Figure 3(b), the first fingerprint at $\ell = 7$ is located at $A_1[3]$. Rather than expending costly memory to store an explicit pointer from $I_0[7]$ to $A_1[3]$, we *dynamically* compute the location by using an operator called $\texttt{rank}(s, i)$, which returns the number of ones in the range $s[1] \ldots s[i]$ in the bit-string s. Our proposed method exploits the fact that the `rank` operator can be efficiently implemented using hardware-optimized instructions that are increasingly available in modern processors. In particular, the `rank`



(a) A bucket.



(b) Rank-indexing.



(c) Packed bucket organization.

Figure 3: Rank-indexed hashing.

operator can be efficiently implemented by combining a bitwise-AND instruction with another operation called $\texttt{popcount}(s)$, which returns the number of ones in the bit-string s. Fortunately, the popcount instruction is becoming increasingly common in modern microprocessors and network processors. For example, current generations of 64-bit x86 processors have this popcount instruction built-in [1, 2]. As we shall see in the evaluation section, very compact constructions can be achieved by setting $L \leq 64$. Since $|I_0| = L$, we can directly compute $\texttt{rank}(I_0, \ell)$ using hardwired 64-bit instructions.

Continuing with the example shown in Figure 3(b), we can compute the location of the first fingerprint at location $\ell = 7$ in A_1 by invoking $a_1 = \texttt{rank}(I_0, 7)$, which will return $a_1 = 3$. We can then match against the fingerprint stored at $A_1[a_1] = A_1[3]$. Similarly, we can check if the chain has ended by checking $I_1[3]$. If $I_1[3] = 1$, then we can locate the next fingerprint in A_2 by computing $a_2 = \texttt{rank}(I_1, 3) = 2$. Given that $I_2[2] = 1$, we can locate the next fingerprint in the chain in A_3 by computing $a_3 = \texttt{rank}(I_2, 2) = 1$. We see that the fingerprint "1010101" is found at $A_3[1]$. Further, we see that the chain has ended since $I_3[1] = 0$. In general, if a chain has extended beyond level j (i.e., $I_j[a_j] = 1$), then the index location to the next subarray A_{j+1} can be simply computed as $a_{j+1} = \texttt{rank}(I_j, a_j)$. Observe that $|I_0| \geq |I_1| \geq \cdots \geq |I_d|$. Therefore, if we use $L \leq 64$, then all rank operations can be directly performed using 64-bit instructions.

2.3.2 Packed Bucket Organization

Thus far above, we provided a high-level description of the idea of rank-indexed hashing. In practice, different buckets will have varying load distributions at the different hash chain locations, which means the number of fingerprints at each level will also fluctuate across buckets. To take advantage of statistical multiplexing, our bucket organization packs the subarrays of fingerprints together into a contiguous array A. This is depicted in Figure 3(c). This way, we can dimension a bucket to store a *fixed* number of Z fingerprints. The Z fingerprint locations can be shared by all the fingerprints in a bucket regardless of their

hash chain location (i.e., regardless of which chain a fingerprint is located). We can also fix the size of the bitmap $I = I_0 I_H$ to $L + Z$ bits with L bits set aside for I_0 and Z bits set aside for $I_H = I_1 I_2 \ldots I_d$. I_0 is referred to as the base index, and I_H is referred to as the higher index. In the example shown in Figure 3(c), L is fixed at 8 and Z is fixed at 12.

Using this packed bucket organization, the size of a subarray A_j and the corresponding bitmap I_j may dynamically change. The current size of a subarray can be readily computed using the `popcount` operator. For example, the size of A_1 and I_1 shown in Figure 3(c) can be computed as `popcount`(I_0). In general, the size of A_{j+1} and I_{j+1} can be computed as `popcount`(I_j).

On insertion or deletion, a fingerprint may be added or removed, respectively (e.g., in the case of a counting Bloom filter, a fingerprint is only removed when the corresponding counter has been decremented to zero). The addition or removal of a fingerprint can increase or decrease the size of a subarray by one. To maintain the packed bucket representation, up to $Z - 1$ fingerprints may have to be shifted in the worst-case. Here again, we can exploit available 64-bit instructions in modern processors to expedite this shifting process. For example, if 8 bits are used to store a remainder, then a 64-bit instruction can shift eight fingerprints at a time. Similarly, for the bitmap I_H, up to $Z - 1$ bits may have to be shifted in the worst-case, but 64 bits can be shifted at a time using 64-bit instructions.

2.3.3 Quasi-Dynamic Bucket Sizing

So far, we have assumed that the fixed number of Z fingerprints allocated to each bucket is chosen to ensure that the actual load of a bucket is less than or equal to Z with high probability. In Table 2.3.3, we consider an example configuration in which each bucket has $L = 64$ hash chain locations and $\lambda = 0.875$. In this case, $\lambda \cdot L = 56$ is also the expected average load. Each table entry indicates the fraction of buckets that are expected to have actual loads greater than the corresponding specified threshold. We see that if we set $Z = 64$, then only 13% of the buckets are expected to overflow. If we set $Z = 74$, then

Table 1: Table entries give the fraction of buckets with actual load greater than some threshold for a bucket with $L = 64$ hash chain locations and $\lambda = 0.875$. Asymptotic tail bounds are provided for three example thresholds.

Threshold	Fraction
Load > 64	0.13
Load > 74	0.01
Load > 128	8e-17

just 1% of the buckets are expected to overflow. And if we set $Z = 128$, then no buckets is expected to overflow with high probability.

Suppose we set $Z = 64$ fingerprint cells. This means that the expected fraction of unused fingerprint cells is just $(64/56) = 1.14$, providing a high degree of space efficiency. However, in this example, we need a way to accommodate the possibility that 13% of buckets might overflow. Conceptually, when a bucket overflows, meaning that all Z fingerprint slots are already occupied on a new insertion, we dynamically *extend* the bucket size by dynamically linking another "chunk" of memory to it. We refer to these chunks of memories as *bucket extensions*. This is illustrated in Figure 4.

To make the example more concrete, suppose $B = 1000$. Then we can statically allocate memory for $J_2 = 0.13B = 130$ second-level buckets. However, these second-level buckets are much smaller than the first-level buckets (and there are fewer number of them). In particular, referring to Table 2.3.3, we see that just 1% of the buckets are expected to have actual loads greater than 74. Suppose we dimension these second-level buckets to hold $Z_2 = 10$ fingerprints. Here, we will use Z_1 rather than Z to indicate the dimensioning of the first-level buckets: i.e., $Z_1 = 64$. Then, for buckets that have been extended to the second-level, $Z = Z_1 + Z_2 = 64 + 10 = 74$ fingerprints are available. In addition, the second-level buckets will provide an additional $Z_2 = 10$ bits for extending the higher index I_H. Only the first-level bucket needs to store the base index I_0. The linkage from a first-level bucket to a second-level bucket can be implemented simply as a memory pointer, which is affordable

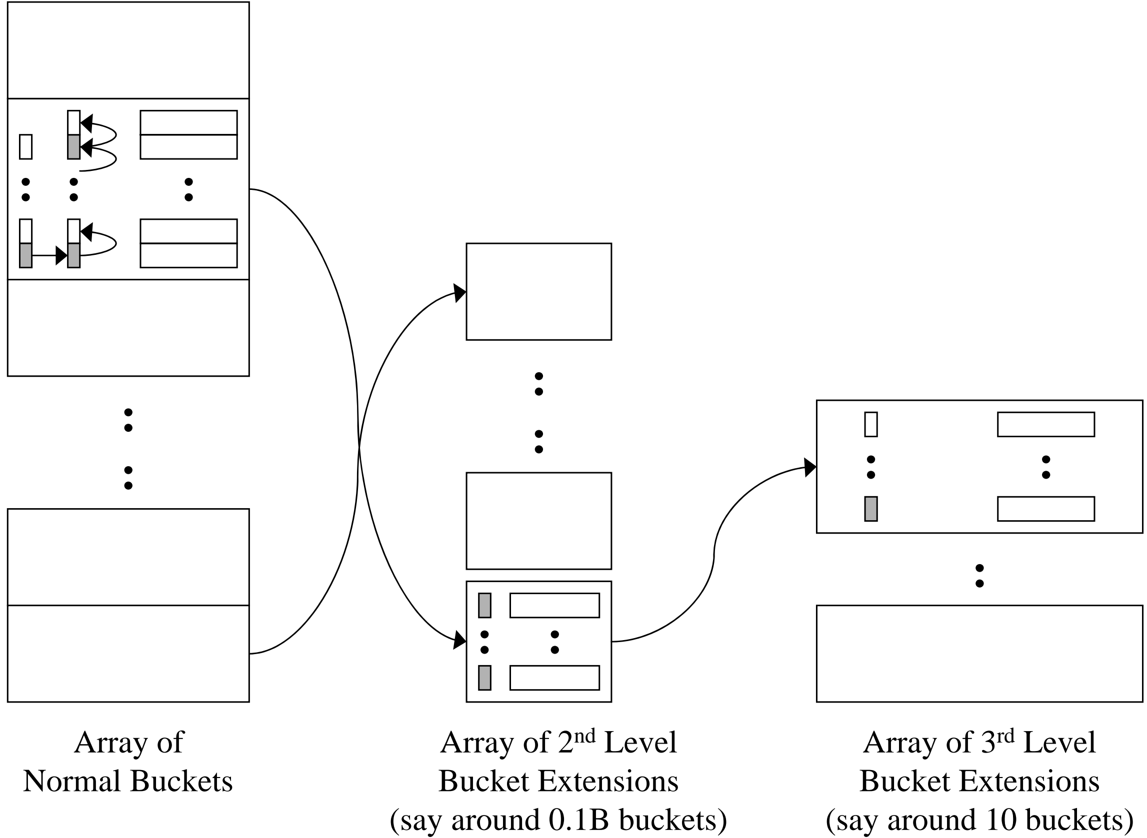


Figure 4: Quasi-dynamic bucket sizing via bucket extensions.

since they are small relative to the size of the buckets, and bucket extensions only occur with a small fraction of buckets. An extension flag can be used to indicate that a bucket has been extended.

With the availability of second-level buckets, with $Z = Z_1 + Z_2 = 74$ fingerprints, then just 1% of buckets are expected to have loads greater than two bucket levels. To accommodate these cases, we can statically allocate memory for $J_3 = 0.01B = 10$ third-level buckets. Suppose we dimension these third-level buckets to provide $Z_3 = 54$ additional fingerprints. Then, for buckets that have been extended to the third-level, $Z = Z_1 + Z_2 + Z_3 = 64 + 10 + 54 = 128$ fingerprints are available. Third-level buckets will also provide an additional $Z_3 = 54$ bits for extending I_H as well. For the parameters shown in Table 2.3.3, as well as in practice, three bucket levels are sufficient, and each bucket level can be dimensioned differently.

Once a bucket has been extended, access to the extended fingerprint array A requires only a small modification. For example, suppose we want to access the fingerprint array location $A[Z_1 + 1]$. Then we simply just use the memory pointer as an offset in computing the memory location.

2.4 Tail Bound Analysis

In this section, we establish a strict tail bound to bound the probability P_o that the bucket extensions for the overflowed buckets are insufficient.

Let random variable $X_i^{(n)}, 1 \leq i \leq B$, denote the number of fingerprints inserted in the i-th bucket. Let O_2 denote the total number of buckets overflowed into the 2nd level bucket extension array, i.e. $O_2 \equiv \sum_{i=1}^{B} 1_{\{X_i^{(n)} > Z_1\}}$. We want to bound the probability that O_2 exceeds the number of 2nd level bucket extensions, i.e. $Pr[O_2 > J_2]$.

In general, we want to bound $Pr[O_l > J_l]$, $l = 2, 3, 4$. Let $W_1 = Z_1$, $W_2 = Z_1 + Z_2$ and $W_3 = Z_1 + Z_2 + Z_3$. Then $O_l = \sum_{i=1}^{B} 1_{\{X_i^{(n)} > W_{l-1}\}}$. Here we define $J_4 \equiv 0$, and $Pr[O_4 > J_4]$ captures the event that the buckets expand out of the 3rd extension. So P_o is bound by

$$Pr[P_o] \leq \sum_{l=2}^{4} Pr[O_l > J_l]$$

In the following we show how to bound $Pr[O_l > J_l]$, and we will use O, W, J instead of O_l, W_l, J_l.

For an analogy, the statistical model for hashing n elements into B buckets is the same as the classical balls-and-bins problem: n balls are thrown independently and uniformly at random into B bins. Then random variable $X_i^{(n)}$ could be translated as the number of balls ended up in the i-th bin after n balls are thrown.

Therefore, $(X_1^{(n)}, \ldots, X_B^{(n)})$ follows the multinomial distribution. The marginal distribution for any $X_i^{(n)}$ is $Binomial(n, \frac{1}{B})$, where $Binomial(N, P)$ denotes the binomial distribution resulting from N trials with success probability P. Since n is large and $\frac{1}{B}$ is small, $X_i^{(n)}$ can be approximated by $Poisson(\frac{n}{B})$, where $Poisson(\lambda)$ denotes the Poisson distribution with

parameter λ.

A naive approach to calculating the probability $\Pr[O > J]$ is to regard $X_i^{(n)}$ as mutually independent. In this case, the sum of the indicator of the events $X_i^{(n)} > Z$ would be under binomial distribution and hence could be easily bounded. However, those random variable $X_i^{(n)}$ are not totally independent since they are conditioned by $\sum_{i=1}^{B} X_i^{(n)} = n$.

Fortunately, we have the following theorem to decouple the weak correlation among $(X_1^{(n)}, \ldots, X_B^{(n)})$ and bound our targeted probability.

Theorem 1. *Let $(X_1^{(n)}, \ldots, X_m^{(n)})$ follow the multinomial distribution of throwing n balls into m bins. Let $(Y_1^{(n)}, \ldots, Y_m^{(n)})$ be independent and identically distributed random variables with distribution $Poisson(\frac{n}{m})$. Let $f(x_1, \ldots, x_m)$ be a nonnegative function which is increasing in each argument separately. Then*

$$E[f(X_1^{(n)}, \ldots, X_m^{(n)})] \leq 2E[f(Y_1^{(n)}, \ldots, Y_m^{(n)})]$$

.

Theorem 1 is derived from [48, Theorem 5.10] using stochastic ordering, which we quote below.

Lemma 1. *Let $(X_1^{(n)}, \ldots, X_m^{(n)})$ and $(Y_1^{(n)}, \ldots, Y_m^{(n)})$ be the same as defined in Theorem 1. Let $f(x_1, \ldots, x_m)$ be a nonnegative function such that $E[f(X_1^{(n)}, \ldots, X_m^{(n)})]$ is monotonically increasing in n. Then*

$$E[f(X_1^{(n)}, \ldots, X_m^{(n)})] \leq 2E[f(Y_1^{(n)}, \ldots, Y_m^{(n)})]$$

.

In practice, we are concerned with the probability of some event A, whose indicator random variable is $f(X_1^{(n)}, \ldots, X_m^{(n)})$, where n is some parameter. So $Pr[A] = E[f(X_1^{(n)}, \ldots, X_m^{(n)})]$. In most cases it is usually intuitive to say that $Pr[A]$ increases as n increases, for example when A is some overflowing event, but it may not be trivial to prove so. On the other hand,

it is usually trivial to see that f is an increasing function. Therefore we introduced the easier-to-use Theorem 1, and we will prove it using stochastic ordering.

Stochastic ordering is a way to compare two random variables. Random variable X is stochastically less than or equal to random variable Y, written $X \leq_{st} Y$, iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. An equivalent definition of $X \leq_{st} Y$ is that $Pr[X > t] \leq Pr[Y > t], -\infty < t < \infty$. The definition involving increasing functions also applies to random vectors $X = (X_1, ..., X_h)$ and $Y = (Y_1, ..., Y_h)$: $X \leq_{st} Y$ iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. Here ϕ is increasing means that it is increasing in each argument separately with other arguments being fixed. This is equivalent to $\phi(X) \leq_{st} \phi(Y)$. Note this definition is a much stronger condition than $Pr[X_1 > t_1, ..., X_h > t_h] \leq Pr[Y_1 > t_1, ..., Y_h > t_h]$ for all $t = (t_1, ..., t_h) \in \mathcal{R}^n$.

Now we state without proof a fact that will be used to prove Proposition 5. Its proof can be found in all books that deal with stochastic ordering [51].

Proposition 1. *Let X and Y be two random variables (or vectors).* $X \leq_{st} Y$ *iff there exists* X' *and* Y' *such that* $\mu(X') = \mu(X)$, $\mu(Y') = \mu(Y)$, *and* $Pr[X' \leq Y'] = 1$. *Here* $\mu(X)$ *means the distribution for X.*

Now we are ready to prove the following proposition.

Proposition 2. *Let* $(X_1^{(n)}, X_2^{(n)}, \ldots, X_m^{(n)})$ *be the same as defined in Theorem 1. For any* $0 \leq n < n'$, *we have*
$(X_1^{(n)}, X_2^{(n)}, \ldots, X_m^{(n)}) \leq_{st} (X_1^{(n')}, X_2^{(n')}, \ldots, X_m^{(n')})$.

Proof. It suffices to prove it for $n' = n + 1$. Our idea is to find random variables Z and W such that Z has the same distribution as $(X_1^{(n)}, X_2^{(n)}, \ldots, X_m^{(n)})$, W has the same distribution as $(X_1^{(n+1)}, X_2^{(n+1)}, \ldots, X_m^{(n+1)})$, and $\Pr[Z \leq W] = 1$. We will use the probability model that is generated by the "throwing $n + 1$ balls into m bins one-by-one" random process. Now given any outcome ω in the probability space Ω, let $Z(\omega) = (Z_1(\omega), Z_2(\omega), ..., Z_m(\omega))$, where $Z_j(\omega)$ is the number of balls in the j_{th} bucket after we throw n balls into these m bins one by

one. Now with all these n balls there, we throw the $(n+1)_{th}$ ball uniformly randomly into one of the bins. We define $W(\omega)$ as $(W_1(\omega), W_2(\omega), ..., W_m(\omega))$, where $W_j(\omega)$ is the number of balls in the j_{th} bin after we throw in the $(n+1)_{th}$ ball. Clearly we have $Z(\omega) \leq W(\omega)$ for any $\omega \in \Omega$ and therefore $\Pr[Z \leq W] = 1$. Finally, we know from the property of the "throwing $n+1$ balls into m bin one-by-one" random process that Z and W have the same distribution as $(X_1^{(n)}, X_2^{(n)}, \ldots, X_m^{(n)})$ and $(X_1^{(n+1)}, X_2^{(n+1)}, \ldots, X_m^{(n+1)})$ respectively. □

Now we are ready to prove Theorem 1.

Proof of Theorem 1. Let $f(x_1, \ldots, x_m)$ be an increasing function. For any $0 \leq n < n'$, we have
$(X_1^{(n)}, X_2^{(n)}, \ldots, X_m^{(n)}) \leq_{st} (X_1^{(n')}, X_2^{(n')}, \ldots, X_m^{(n')})$ by Proposition 5. By definition of stochastic ordering, we have $E[f(X_1^{(n)}, \ldots, X_m^{(n)})] \leq E[f(X_1^{(n')}, \ldots, X_m^{(n')})]$. Therefore $E[f(X_1^{(n)}, \ldots, X_m^{(n)})]$ is monotonically increasing in n, and the theorem follows from Lemma 1.

□

Considering the following function

$$f(x_1, \ldots, x_B) = 1_{\{(\sum_{i=1}^{B} 1_{x_i > W}) > J\}}$$

The targeted probability we want to bound, $\Pr[O > J]$, could be regarded as the expectation of $f(X_1^{(n)}, \ldots, X_B^{(n)})$, i.e.

$$\Pr[O > J] = E[f(X_1^{(n)}, \ldots, X_B^{(n)})].$$

$f(x_1, \ldots, x_B)$ is obviously an increasing function. Therefore $E[f(X_1, \ldots, X_B)] \leq 2E[f(Y_1, \ldots, Y_B)]$, thanks to Theorem 1.

Y_i's are distributed as $Poisson(\frac{n}{B})$. Therefore $Pr[Y_i > W] = Poissontail(\frac{n}{B}, W)$, where $Poissontail(\lambda, K)$ denotes the tail probability $Pr[Y > K]$ where Y has distribution $Poisson(\lambda)$.

$E[f(Y_1, \ldots, Y_B)]$ denotes the probability that more than J of the events $Y_i > W$ happen. Since these events are mutually independent, the probability is $Binotail(B, Poissontail(\frac{n}{B}, W), J)$,

where $Binotail(N, P, K)$ denotes the tail probability $Pr[Z > K]$ where Z has distribution $Binomial(N, P)$.

So we get

$$Pr[O > J] \leq 2Binotail(B, Poissontail(\frac{n}{B}, W), J). \quad (1)$$

$Pr[O_4 > J_4]$ is a special case since $J_4 = 0$. We do not need theorem 1, and we have a better bound

$$\begin{aligned} Pr[O_4 > 0] &= \Pr[(\max_i X_i^{(n)}) > W_3] \\ &\leq \sum_{i=1}^{B} \Pr[X_i^{(n)} > W_3] \\ &= B \times Binotail(n, \frac{1}{B}, W_3) \end{aligned}$$

2.5 Evaluation

In this section, we present a set of numerical results computed from the tail bound theorems that are derived in Section 2.4 and the appendix. For any targeted false positive probability ϵ, we apply the tail bound theorems to derive optimal configurations under different constraints.

One constraint we impose is on the parameter L. Ideally, larger buckets would lead to better statistical multiplexing and better space savings. However, we constrain L to be at most 64 to ensure that the `rank` and `popcount` operations described in Section 2.3 can be directly implemented using hardware-optimized 64-bit instructions that are readily available in modern microprocessors and network processors. For example, current generations of 64-bit $x86$ processors support such operations very efficiently [1, 2].

In Table 2, we present representative sizing results for rank-indexed hashing under different false positive probabilities. These sizing results assume $n = 100,000$ elements and a probability of $P_o = 10^{-10}$ that the bucket sizing is not enough to store new fingerprints.

Table 2: Representative results for rank-indexed hashing under various parameter settings ($P_o = 10^{-10}$, $n = 10^5$).

ϵ	λ	R	L	Z_1	Z_2	Z_3	J_2/B	J_3/B
1%	0.64	6 bits	60	45	8	45	17.9%	2.7%
0.1%	0.92	10 bits	64	63	17	50	36.0%	1.7%
0.01%	0.86	13 bits	61	59	13	48	23.3%	1.8%

In particular, the results presented are for false positive probabilities of $\epsilon = 1\%$, 0.1%, and 0.01%. Given the different parameter settings, we apply our tail bounds to optimize the configurations to minimize storage cost. The derived configurations are in terms of the load factor λ, the number of hash chain locations per bucket L, and the number of allocated entries in the bucket and bucket extensions Z_1, Z_2, Z_3.

For a configuration of these parameters, the amount of memory required is determined as follows:

$$\begin{aligned}
\mathcal{S}_1 &= (L + Z_1) + Z_1 r + (1 + \lfloor \log J_2 \rfloor) \\
\mathcal{S}_2 &= 1 + Z_2 + Z_2 r + (1 + \lfloor \log J_3 \rfloor) \\
\mathcal{S}_3 &= 1 + Z_3 + Z_3 r \\
\mathcal{S} &= B\mathcal{S}_1 + J_2\mathcal{S}_2 + J_3\mathcal{S}_3
\end{aligned}$$

Here, $\mathcal{S}_1$ is the storage requirement of one normal (first-level) bucket. The three additive components correspond to the bitmap index, the fingerprints, and the pointer to bucket extensions. (J_2 is the number of pre-allocated second-level bucket extensions). $\mathcal{S}_2$ and $\mathcal{S}_3$ are the size of the second-level and third-level bucket extensions. The one bit is for indicating occupancy. Then the total memory cost $\mathcal{S}$ would be the sum of the storage requirements for the B normal buckets, J_2 second-level bucket extensions and J_3 third-level bucket extensions. Then the amortized storage cost per element can be computed as $\frac{\mathcal{S}}{n}$.

In Table 3 and Table 4, we compare results for the standard Bloom filter function [5]

and for the counting Bloom filter function [22]. In addition to comparing with the standard Bloom filter constructions, we also compare with a recently proposed fingerprint hash table construction called d-left hashing [7, 6]. The standard Bloom filter functionality results are shown in Table 3. In comparison to the d-left hashing construction, our rank-indexed hashing results outperform it in storage cost by 23.2% to 29.5%. In comparison to the standard Bloom filter construction, our rank-indexed hashing construction is able to outperform a standard Bloom filter construction in storage cost for false positive probabilities at or below just 0.1%. This is significant since a standard Bloom filter construction is widely regarded as a very space-efficient data structure for approximate membership query problems, and it is often desirable to have a false positive probability smaller than 0.1% in many applications.

Table 4 present the counting Bloom filter functionality results. In comparison to a standard counting Bloom filter construction, our rank-indexed hashing construction is able to outperform in storage cost by a factor of three for a false positive probability of just 1%, and it is able to outperform the d-left hashing construction 22% to 27% at the same false positive probabilities.

In table 5, we present another advantage of Rank-Indexed Bloom Filter, which could save more space after "compression". When used for transferring, there is no need to send the vacant entries at the end of the higher index array and the fingerprint array in each bucket. After removing all these vacant entries, the trimmed size is only $S_p = 1/\lambda + (r + 1)$ bits per item. It is sufficient to transfer these trimmed buckets without any additional information, since the boundary of each array could be determined by counting the index bitmaps. The trimmed size is smaller than the optimized standard Bloom Filter size. To use Compressed Bloom Filter to achieve the same compression effect, much larger original Bloom Filter size will be needed. In table 5, we could see that our Rank-Indexed scheme could easily achieve very compact trimmed size, while a much larger original array is needed for a Compressed Bloom Filter to achieve the same compression ratio.

Table 3: Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the standard Bloom filter function.

ϵ	standard	d-left	Rank	Comparison	
				vs. standard	*vs.* d-left
1%	9.6	15.0	10.6	+10.1%	−29.5%
0.1%	14.6	18.0	14.4	−1.1%	−26.3%
0.01%	19.1	22.2	18.2	−4.4%	−23.2%

Table 4: Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the counting Bloom filter (CBF) function.

ϵ	standard CBF	d-left CBF	Rank CBF	Comparison	
				vs. standard	*vs.* d-left
1%	38.3	17.6	13.0	−66%	−27%
0.1%	58.4	22.3	16.8	−71%	−24%
0.01%	76.3	26.4	20.6	−73%	−22%

Since d-left Bloom Filter is also fingerprint hash-table based, it could also be "compressed" similarly. However, its compression effect is much weaker, as also presented in table 5.

Finally, storage cost comparisons are presented in Figure 5 and Figure 6 for false positive probabilities ranging from 10^{-1} to 10^{-7}. As shown in these plots, the proposed rank-indexing approach performs very competitively over this entire range.

2.6 *Combination with* d*-left Hashing*

As briefly introduced in Section 2.2.2, d-left hashing [63] is an efficient way to construct Hash Table. It has been proposed to use d-left hashing to construct fingerprint-hash-table-based Bloom Filter, called d-left Counting Bloom Filter [6]. However, in Section 2.2.2 we have provided a typical construction of d-left Bloom Filter and shown that it needs considerable extra space. In this section, we will first generalize the analysis of memory utilization of d-left Bloom Filter in Section 2.6.1, which will help readers to understand the motivation of combining our rank-indexing with d-left Hashing. Then we will present the Rank-Indexed d-left Bloom Filter in Section 2.6.2, and the evaluations of the combined

Table 5: Comparison of transferring cost (in bits) per element against Compressed Bloom Filter and *d*-left Bloom Filter

ϵ	Scheme	Packed Size	Uncompressed Compression BF[1]	Compression ratio[2]
1%	Rank	8.6	30	89%
1%	*d*-left	11.7	−−[3]	122%
0.1%	Rank	12.1	150	83%
0.1%	*d*-left	15.2	−−[3]	104%
0.01%	Rank	15.2	1.3×10^3	79%
0.01%	*d*-left	18.3	27	96%

[1] "Uncompressed Compression BF" is the size of the uncompressed original array of a Compressed Bloom Filter to achieve the same compression ratio.
[2] "Compression ratio" is the ratio of the trimmed size of a Rank-Indexed or *d*-left scheme vs. the size of an optimized standard Bloom Filter, under the same false positive rate.
[3] Omitted since the trimmed size is still larger than the size of a corresponding optimized standard Bloom Filter under the same false positive rate.

scheme in Section 2.6.3.

2.6.1 Analysis of Memory Utilization of *d*-left Bloom Filter

The typical construction of *d*-left Bloom Filter given in Section 2.2.2 and also in [6] is by 4 tables of buckets, 8 fingerprints per bucket and an average load of 6 fingerprints per bucket. Suppose each fingerprint has x bits, the false positive probability would be:

$$\epsilon = 4 \times 6 \times 2^{-x}$$

The total storage per inserted item would be

$$\frac{8}{6} \times x$$

In short, $\frac{8}{6}(\log_2(1/\epsilon) + \log_2(4 \times 6))$ bits of storage per inserted item would be needed by this typical construction. Would other settings of parameters improve the efficiency of storage? The paper [6] doesn't provide any reason for selecting the parameters above. However we could calculate by ourselves.

Suppose we have d tables, average load of m items per bucket , n items to be inserted in total, we need $m + \delta(d, m, n)$ per bucket to guarantee an extremely small probability of

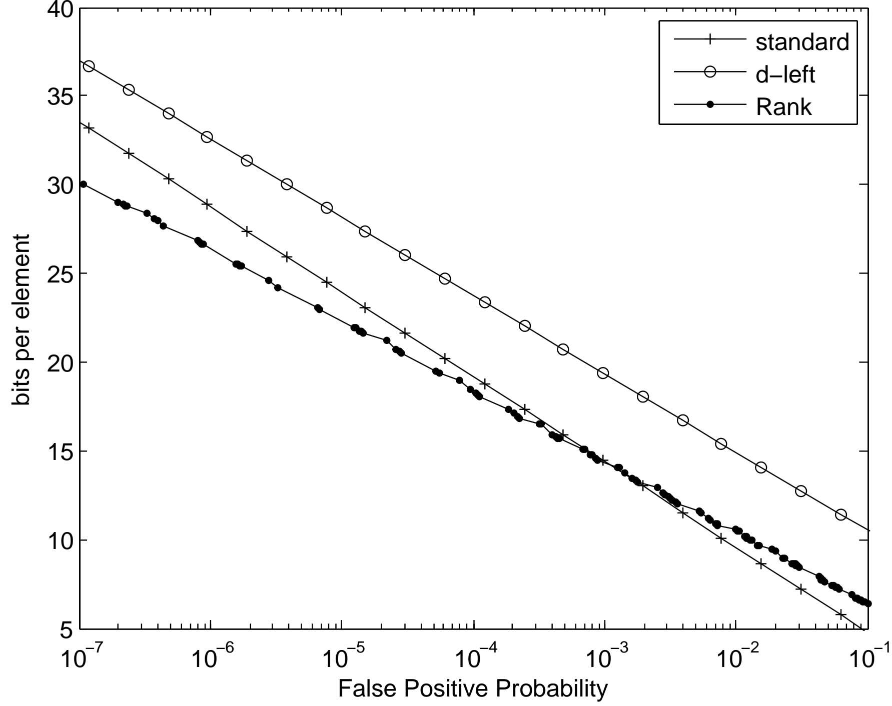


Figure 5: Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the standard Bloom filter function.

bucket overflow, since there is no backup mechanism for bucket overflow in *d*-left Bloom Filter. The memory needed per inserted item would be

$$(1 + \frac{\delta(d, m, n)}{m})(\log_2(1/\epsilon) + \log_2 md) \tag{2}$$

Unfortunately, there hasn't been found any way to provide a solid and precise statistical guarantee for $\delta(d, m, n)$.

In [8] Broder et al provides a fluid-model based method to numerically approximately calculate the overflow probability. The nature of the method is to use differentiate equations to "simulate" the process of insertion and get the expected fraction of buckets that have more than an arbitrary number of elements inserted. In detail, it is to solve the following equations

$$\frac{dx_{i,j}}{dt} = (x_{i-1,j} - x_{i,j}) \prod_{k=j+1}^{d} x_{i-1,k} \prod_{k=1}^{j-1} x_{i,k}$$

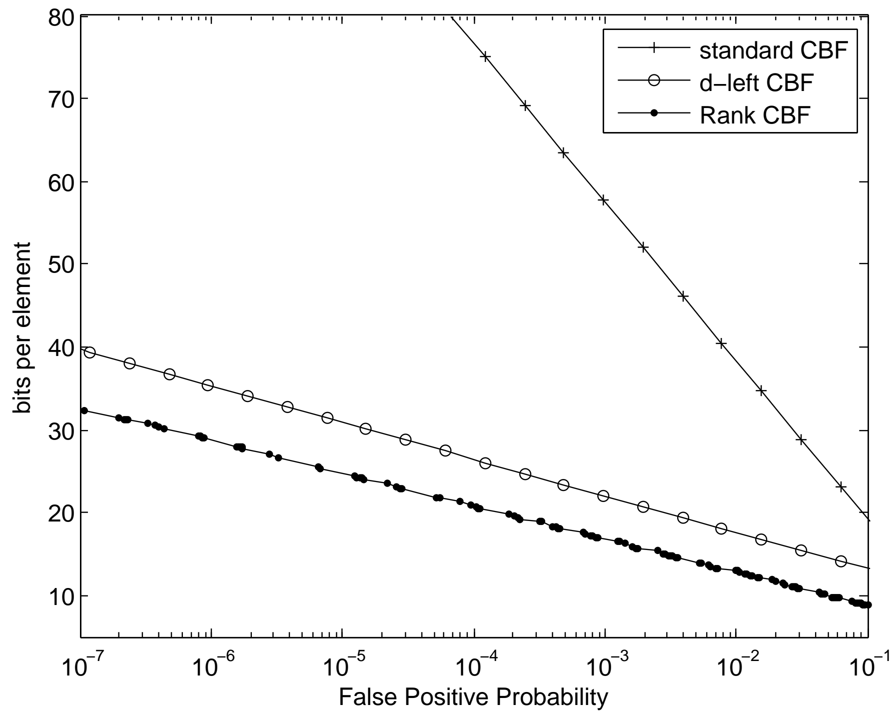


Figure 6: Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the counting Bloom filter (CBF) function.

with boundary conditions:

$$x_{0,j}(t) = 1, \text{ for } j = 1, ..., d$$

$$x_{k,j}(0) = 0, \text{ for } j = 1, ..., d; k > 0$$

The solution $x_{i,j}(t)$ would be the expected fraction of buckets in the jth table that have at least i items , when Bt items have been inserted into a d-left hash table of B buckets in each table.

We solve the equations above and summarize the results as the following: if we use two choices (two tables), no matter how large m is, δ needs to be 4 to achieve an overflow fraction lower than 10^{-20}, i.e. we need 4 more entries per bucket to overcome fluctuation. If we use three choices, δ is 3 . If we use four choices,δ is 2 . This result coincides with Vocking's asymptotical bound $\delta(d, m, n) = \frac{\ln \ln n}{d\phi_d}$ [63], where is asymptotically unrelated to the particular value of m, which means buckets need almost the same size of extra space to overcome fluctuations, when only the bucket size is different.

Intuitively, since $\delta(d, m, n)$ is almost a constant on m, it would benefit the memory efficiency a lot if we enlarge the bucket size in d-left Bloom Filter. However, this intuition is wrong.

In Figure 7, we summarize the relationship between the expected load per bucket and the memory efficiency. We could observe that increasing bucket size from 8 items to 10 items could help a little bit and save around 0.5 bits per item, while still around 5 bits more compared to the standard Bloom Filter (which is 11.5 bits per item for the false probability of 0.4%). However, increasing the average load per bucket even more would help very little. The reason is that, although large bucket would lead to relatively small extra space per bucket, it would also increase the $\log_2 md$ term in (2). In another word, larger bucket size would not only increase the cost to access each fingerprint, but also contribute to more false positives. Hence the 4-table-8-items-per-bucket design is almost the best for d-left Bloom Filter. In one word, enlarging bucket size in d-left Bloom Filter could only improve memory efficiency very little. For this reason, we resort to combining rank-indexing into

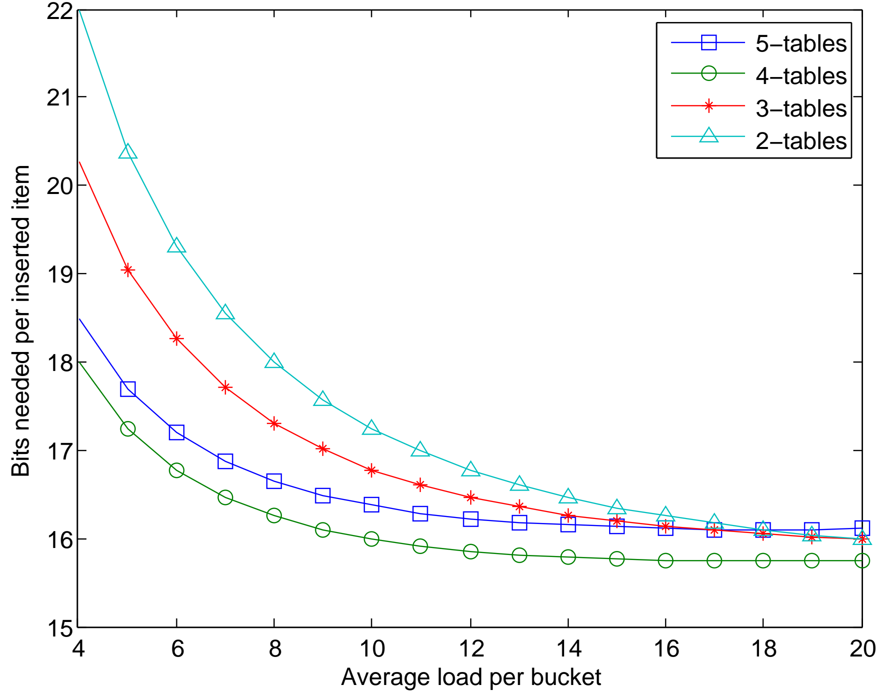


Figure 7: memory savings through adjusting the size of bucket in d-left Bloom Filter, when targeted false positive rate is 0.4%

d-left Bloom Filter.

2.6.2 Rank-Indexed *d*-left Bloom Filter

Although larger bucket might be a bad idea for a pure *d*-left Bloom Filter, rank-indexed hashing would make some difference. The key idea of Rank-Indexed *d*-left Bloom Filter is to use relatively larger bucket (say, 64) and then uses rank-indexing inside the bucket.

An illustration of Rank-Indexed *d*-left Bloom Filter is presented in Figure 8. Now we have two tables. Each table has B buckets and hence $2B$ buckets in total. The structure inside each bucket is the same as the rank-indexed bucket presented in Section 2.3. Thanks to *d*-left hashing, here we no longer need the second and third level of bucket arrays since the *d*-left hashing already guarantees a small overflow probability (although the guarantee is theoretically less precise). Also thanks to rank-indexed hashing, we could have a larger bucket that could easily support query and insertion and we also only need two tables

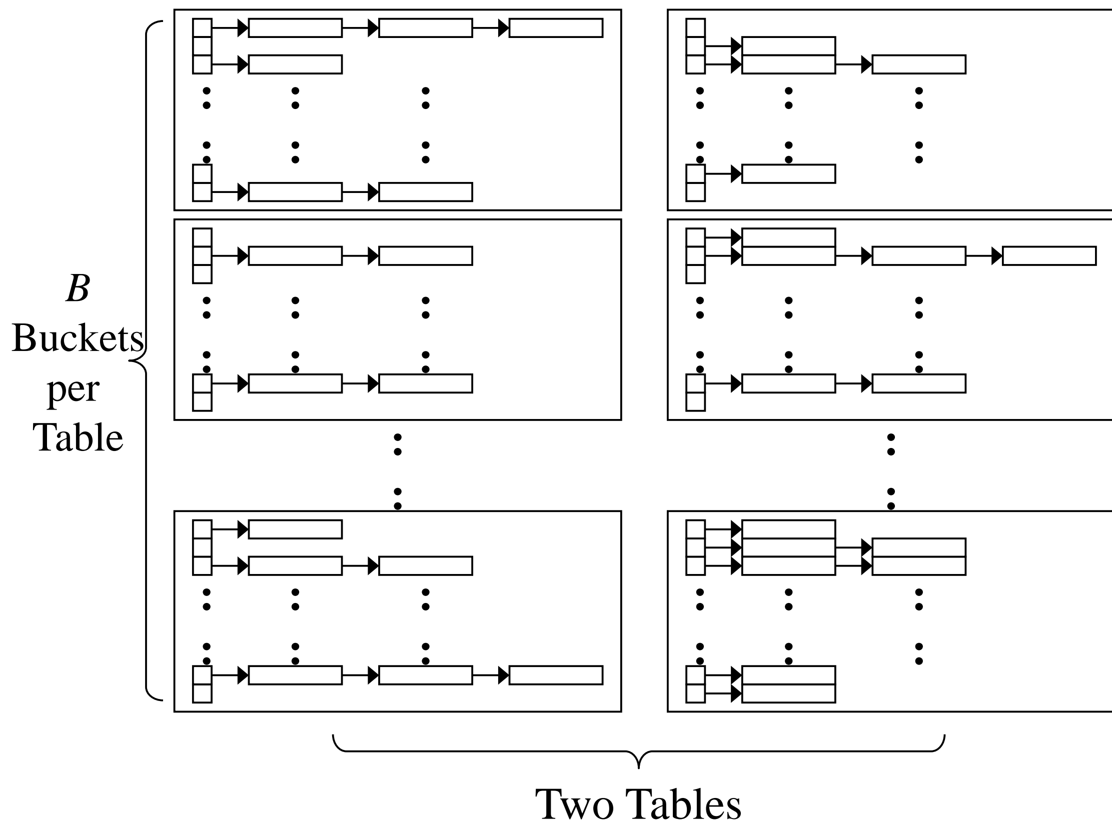


Figure 8: An illustration of *d*-left Bloom Filter

instead of four tables, which would reduce false positives and memory accesses.

The insertion and deletion procedures are multi-inherited from the original d-left Bloom Filter and the original rank-indexed Bloom Filter. The procedure of selecting buckets to be inserted or scanned is the same as the original d-left Bloom Filter, while the the operations inside one bucket is the same as the original rank-indexed Bloom Filter.

The insertion procedure is as follows. For each element to be inserted, we still use a single hash function $H : U \rightarrow [B] \times [L] \times [R]$ that hashes the element in S into three parts $H(x) = (b, \ell, r)$. Due to the d-left structure, we also need another random permutation function $H : [B] \times [L] \times [R] \rightarrow [B] \times [L] \times [R]$ to map $H(x)$ to another value $H_p(H(x)) = (b_p, \ell_p, r_p)$. (please refer to [7] for the reason of this random permutation.) The next step is to check two buckets, the bth bucket in the first table and b_p^{th} bucket in the second table, and select the one with the least occupancy to be inserted in. If two buckets have the same total number of fingerprints inserted, select the one in the first table. In the end, insert the fingerprints into the ℓth (or ℓ_pth if the second table is selected) chain within the selected bucket. The d-left hashing principle guarantees that the probability that the bucket has no place to be inserted would be extremely small.

The query and deletion procedures are very similar, and hence omitted to save space.

2.6.3 Evaluation and Discussion of Rank-Indexed *d*-left Bloom Filter

According to the result calculated based on the procedures presented in [8] and summarized in Section 2.6.1, if we plan to have x fingerprints on average in one bucket, we only need $x+4$ pre-allocated locations of fingerprints. Hence we could have the following representative results presented in Table 6. In those representative results, we choose to select the same Z as L, since this would enhance memory alignment.

In Table 6, $\mathcal{S}_1$ denotes the number of bits per bucket, $\mathcal{S}/n$ denotes the average number of bits needed per inserted item. Those two are calculated based on the following formulas.

Table 6: Representative results for rank-indexed *d*-left hashing Bloom Filter .

ϵ	λ	r	L	Z	λL	$\mathcal{S}_1$	$\mathcal{S}/n$
1.5%	0.94	6 bits	64	64	60	512	8.5
1.4%	0.88	6 bits	32	32	28	256	9.1
0.09%	0.94	10 bits	64	64	60	768	12.8
0.09%	0.88	10 bits	32	32	28	384	13.7
0.011%	0.94	13 bits	64	64	60	960	16
0.011%	0.88	13 bits	64	32	60	480	17.1

Table 7: Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the standard Bloom filter function.

ϵ	*d*-left	Rank	Ranked *d*-left	Comparison	
				vs. *d*-left	*vs.* rank
1.5%	14.2	9.8	8.5	-40%	-11%
0.09%	19.6	15.5	12.8	-34%	-17%
0.011%	23.6	19.2	16	-32%	-17%

We also compare the results of the rank-indexed *d*-left Bloom Filter against to the original rank-indexed hashing Bloom Filter and the original *d*-left hashing Bloom Filter in Table 8 and Table 7.

We could see that, after combining both rank-indexed hashing and *d*-left hashing together, we could even greatly save the memory usage. However, we should keep in mind that *d*-left hashing is saving memory cost in cost of less memory locality (lookup at least two buckets instead of one) and increased amortized access cost.

Table 8: Comparisons of storage cost (in bits) per element to achieve the same false positive probability ϵ for the counting Bloom filter (CBF) function.

ϵ	*d*-left CBF	Rank CBF	Ranked *d*-left CBF	Comparison	
				vs. *d*-left	*vs.* rank
1%	17.7	13.2	10.7	– 49 %	– 19 %
0.09%	24.5	18.3	14.9	– 39%	– 19 %
0.011%	29.5	23.0	18.1	– 39 %	– 21 %

2.7 Conclusion

In this chapter, we have described a new fingerprint hash table construction that can achieve the same functionalities as Bloom filters, counting Bloom filters, and other variants. The construction is based on a new method called Rank-Indexed Hashing that can achieve very compact representations. We have provided analysis and numerical evaluations to show the storage performance of the proposed approach. In particular, a rank-indexed hashing construction that offers the same functionality as a counting Bloom filter can be achieved with a factor of three or more in space savings even for a false positive probability of just 1%. Even for a basic Bloom filter function that only supports membership queries, a rank-indexed hashing construction requires less space for a false positive probability as high as 0.1%, which is significant since a standard Bloom filter construction is widely regarded as extremely space-efficient for approximate membership problems.

CHAPTER III

BRICK: RANK-INDEXING TECHNIQUE FOR EXACT ACTIVE STATISTICS COUNTER ARRAY ARCHITECTURE

3.1 Problem Overview

It is widely accepted that network measurement is essential for the monitoring and control of large networks. For implementing various network measurement, router management, and data streaming algorithms, there is often a need to maintain very large arrays of statistics counters at wirespeeds (e.g., million counters for per-flow measurements). For example, on a 40 Gb/s OC-768 link, a new packet can arrive every 8 ns and the corresponding counter updates need to be completed within this time. While implementing large counter arrays in SRAM can satisfy performance needs, the amount of SRAM required for worst-case counter sizes is often both infeasible and impractical. Therefore, researchers have actively sought alternative ways to realize large arrays of statistics counters at wirespeeds [56, 53, 55, 70].

In particular, several SRAM-efficient designs of large counter arrays based on hybrid SRAM/DRAM counter architectures have been proposed. Their baseline idea is to store some lower order bits (e.g., 9 bits) of each counter in SRAM, and all its bits (e.g., 64 bits) in DRAM. The increments are made only to these SRAM counters, and when the values of SRAM counters become close to overflow, they will be scheduled to be "committed" back to the corresponding DRAM counter. These schemes all significantly reduce the SRAM cost. For example, the scheme by Zhao et al. [70] achieves the theoretically minimum SRAM cost of between 4 to 6 bits per counter, when the speed difference between SRAM and DRAM ranges between 10 (50ns/5ns) and 50 (100ns/2ns). However, in these schemes, while writes can be done as fast as on-chip SRAM latencies (2 to 5ns), read accesses can

only be done as slowly as DRAM latencies (e.g., 60 to 100ns). Therefore, such schemes only solve the problem of so-called *passive counters* in which full counter values in general do not need to be read out frequently (not until the end of a measurement epoch). Besides the problem of slow reads, hybrid architectures also suffer from the problem of significantly increasing the amount of traffic between SRAM (usually on-chip) and DRAM (usually off-chip) across the system bus. This may become a serious concern in today's network processors, where system bus and DRAM bandwidth are already heavily utilized for other packet processing functions [70].

While passive counters are good enough for many network monitoring applications, a number of other applications require the maintenance of *active counters*, in which the values of counters may need to be read out as frequently as they are incremented, typically on a per packet basis. In many network data streaming algorithms [15, 21, 39, 40, 28, 67, 69], upon the arrival of each packet, values need to be read out from some counters to decide on actions that need to be taken. For example, if Count-Min sketch [15] is used for elephant detection, we need to read the counter values on a per packet basis because such readings will decide whether a flow needs to be inserted into a priority queue (implemented as a heap) that stores "candidate elephants". A prior work on approximate active counters [58] identifies several other data streaming algorithms that need to maintain active counters, including multistage filters for elephant detection [21] and online hierarchical heavy hitter identification [67]. Currently, all existing algorithms that use active counters implement them as full-size SRAM counters. An efficient solution for exact active counters clearly will save memory cost for all such applications.

3.1.1 Our approach and contributions

In this dissertation work, we propose the first solution to the open problem of how to efficiently maintain exact active counters. Our objective is to design an exact counter array scheme that allows for extremely fast read *and* write accesses (at on-chip SRAM speeds).

However, these goals will clearly push us back to the origins of using an array of full-size counters in SRAM if we do not impose any additional constraint on the counter values. Fast read access demands that the counters reside entirely in SRAM and we can make the values of each counter large enough (and random enough) so that each of them needs the worst-case (full-size) counter size. Therefore we will solve our problem under a very natural and reasonable constraint. We assume that the total number of increments, which is exactly the sum of counter values in the array, is bounded by a constant M during the measurement interval.

This constraint is a reasonable constraint for several reasons. First, this constraint is natural since the number of increments is bounded by the maximum packet arrival rate times the length of the measurement epoch. We can easily enforce an overall count sum limit by limiting the length of the measurement epoch. Moreover, this constraint has been assumed in designing other memory-efficient data structures such as Spectral Bloom Filters [14]. Furthermore, our scheme will work for arbitrarily large M values, although its relative memory savings compared to full-size counters get gradually lower with larger M values.

Let N be the total number of counters in the array. Then the ratio $\frac{M}{N}$ corresponds to the (worst-case) average value of a counter, which is indeed a more relevant parameter than M for evaluation purposes, as it corresponds to the "per-counter workload". We observe that small $\frac{M}{N}$ ratio is dictated by many real-world applications. For example, if we use a Count-Min [15] sketch with $\ln\frac{1}{\delta}$ arrays of $\frac{e}{\epsilon}$ ($e \approx 2.718$) counters each, for estimating the sizes of TCP/UDP flows, then with probability at least $1-\delta$, the CM-sketch overcounts (it never under-counts) by at most $M\epsilon$. Suppose we set δ to 0.1 and ϵ to 10^{-5} so that we use a total of $\ln(\frac{1}{0.1})\times\frac{e}{10^{-5}} \approx 6.259\times10^5$ counters. When the total number of increments M is set to 10^8 and correspondingly the average counts per counter $\frac{M}{N}$ is approximately 160, we can guarantee that the error is no more than 1,000 (= $10^8\times10^{-5}$) with probability at least 0.9. However, 1,000 are considered very large errors and hence for practice we always want $\frac{M}{N}$

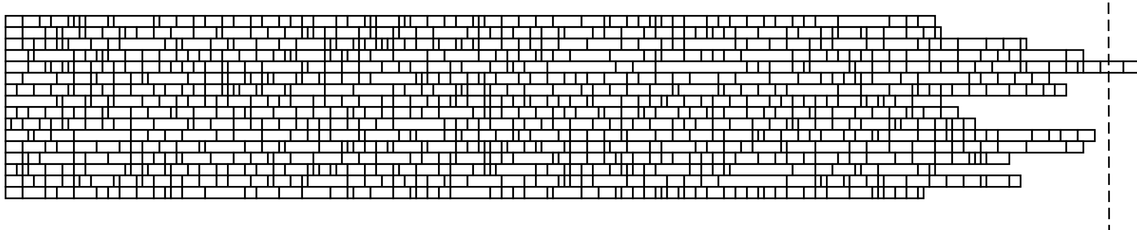


Figure 9: BRICK wall (conceptual baseline scheme)

to be much smaller.

We emphasize that even when the ratio $\frac{M}{N}$ is small, it is still important to figure out ways to save memory, as naive implementations can be grossly wasteful. For example, let the total counts be $M = 16$ million and the number of counters be $N = 1$ million. In other words, the average counter value $\frac{M}{N}$ is 16. Since all increments can go to the same counter, fixed-counter-size design would require a conservative counter size of $\lg(16 \times 10^6) = 24$ bits. However, as we will show, our scheme can significantly reduce the SRAM requirement, which is very important for ASIC implementations where SRAM cost is among the primary costs.

In this chapter, we present an exact active counter architecture called Bucketized (B) Rank (R) Indexed (I) Counter (CK), or BRICK. It is built entirely in SRAM so that both read and increment accesses can be processed at tens to hundreds of millions of packets per second. In addition, since it is stored entirely in SRAM, it will not introduce traffic between SRAM and DRAM. This makes it also a very attractive solution for passive counting applications in which the aforementioned problem of increased traffic over system bus caused by the hybrid SRAM/DRAM architecture becomes a serious concern.

The basic idea of our scheme is intuitive and is based on a very familiar networking concept: statistical multiplexing. Our idea is to bundle groups of a fixed number (let it be 64 in this case) of counters, which is randomly selected from the array, into buckets. We allocate just enough bits to each counter in the sense that if its current value is C_i, we allocate $\lfloor \log_2 C_i \rfloor + 1$ bits to it. Therefore, counters inside a bucket have variable widths. Suppose the mean width of a counter averaged over the entire array is γ. By the law of

large numbers, the total widths of counters in most of the buckets will be fairly close to γ multiplied by the number of counters per bucket. Depicting each counter as a "brick", as shown in Figure 9, a section of the "brick wall" illustrates the effect of statistical multiplexing, where each horizontal layer of bricks (consisting of 64 of them) corresponds to a bucket and the length of bricks corresponds to the real counter widths encoding flow sizes in a real-world Internet packet trace (the USC trace in Section 3.6.2).

As we see in this figure, when we set the bucket size to be slightly longer than 64γ (the vertical dashed line), the probability of the total widths of the bricks overflowing this line is quite small; among the 20 buckets shown, only 1 of them has an overflow. Although overflowed buckets need to be handled separately and will cost more memory, we can make this probability small and the overall overflow cost is small and bounded. Therefore, our memory consumption only needs to be slightly larger than 64γ per bucket.

This baseline approach is hard to implement in hardware in practice for two reasons. First, we need to be able to randomly access (i.e., jump to) any counter with ease. Since counters are of variable sizes, we still need to spend several bits per counter for the indexing within the bucket. Note that being able to randomly access is different from being able to delimit all these counters. The latter can be solved with by prefix-free coding (e.g., Huffman coding [16]) of the counter values. Those coding techniques would replace the counter values with variable-length symbols, which could make the size of storage much smaller while the overhead of accessing and modifying data much larger.

BRICK addresses these two difficulties with a little more overall SRAM cost. It allows for very efficient read and expansion (for increments that increase the width of a counter such as from 15 to 16). A key technique in our data structure is an indexing scheme called *rank indexing*, borrowed from the compression techniques in [35, 18, 59, 33]. The operations involved in reading and updating this data structure are not only simple for ASIC implementations, but are also supported in modern processors through built-in instructions such as "shift" and "popcount" so that software implementation is efficient (as the involved

basic operations such as shift and popcount are supported by modern processors [1, 2]). Therefore our scheme can be implemented efficiently both in hardware or software.

3.2 Background and Related works

In this section, we compare and contrast our work with previous approaches. One category of approaches is based on the idea of a SRAM/DRAM hybrid architecture [56, 53, 55, 70]. The state of art scheme [70] only requires $\log_2 \mu$ bits per counter where μ is the speed different between SRAM and DRAM. This translates into between 4 to 6 SRAM bits per SRAM counter. However, the read can take quite long (say at least 100ns). Therefore, these approaches only solve the passive counting problem.

Another category of approaches is existing active counter solutions [49, 17, 58], which are all based on the approximate counting idea invented by Morris [49]. The idea is to probabilistically increment a counter based on the current counter value. However, approximate counting in general has a very large error margin when the number of bits used is small because the possible estimation values are very sparsely distributed in the range of possible counts. Therefore, when the counter values are small (say 5), its estimation can have a very high relative error (well over 100%). This is not acceptable in network accounting and data streaming applications where small counter values can be important for overall measurement accuracy. In fact, when the (worst-case) average counter value $\frac{M}{N}$ is no more than 128, the SRAM cost of our BRICK scheme (about 12 bits) is no more than that of [58], which is approximate.

Recently, another counter architecture called counter braids [45] has been proposed, which is inspired by the construction of LDPC codes [26] and can keep track of exact counts of all flows without remembering the association between flows and counters. At each packet arrival, counter increments can be performed quickly by hashing the flow label to several counters and incrementing them. The counter values can be viewed as a linear transformation of flow counts, where the transformation matrix is the result of hashing *all*

flow labels during a measurement epoch. However, counter braids are not active and are in fact "more passive" than the SRAM/DRAM hybrid architectures. To find out the size of a single flow, one needs to decode all the flow counts through a fairly long iterative decoding procedure. [1].

Finally, Spectral Bloom Filter [14] has been proposed, which provides an internal data structure for storing variable width counters. It uses a hierarchical indexing structure to locate counters that are packed next to each other, which allows for fast random accesses (reads). However, an update that causes the width of the counter i to grow will cause a shift to counters $i + 1$, $i + 2$, ..., which can have a global cascading effect even with some slack bits provided in between, making it prohibitively expensive when there can be millions of counters. As acknowledged in [14], although the expected amortized cost per update remains constant, and the global cascading effect is small in the average case, the worst-case cannot be tightly bounded. Therefore, SBF with variable width encoding is not an active counter solution as it cannot ensure fast per-packet write accesses at every packet arrival, forcing it to become a mostly-read-only data structure in the sense that updates should be orders of magnitude less frequent than queries.

3.3 Design of BRICK

In this section, we describe the proposed BRICK counter architecture. The objective of BRICK is to efficiently encode a set of N *exact active* counters $C_1, C_2, \ldots, C_N$, under the constraint that throughout a network measurement epoch the total counts[2] across all counters $\sum_{i=1}^{N} C_i$ is no more than a pre-determined threshold M, which is carefully justified in Section 3.1. As we explained earlier, since all increments can go to the same counter, the value of a counter can be as large as M, and hence the worst-case counter width is $L = \lfloor \log_2 M \rfloor + 1$. However, it is unnecessarily expensive to allocate L bits to every counter

[1] In [45], they need 25 seconds on a 2.6GHz computer to decode the flow counts inside a 6-minute-long traffic trace.

[2] Here with an abuse of notation, we will use C_i to denote both the counter and its current count (value).

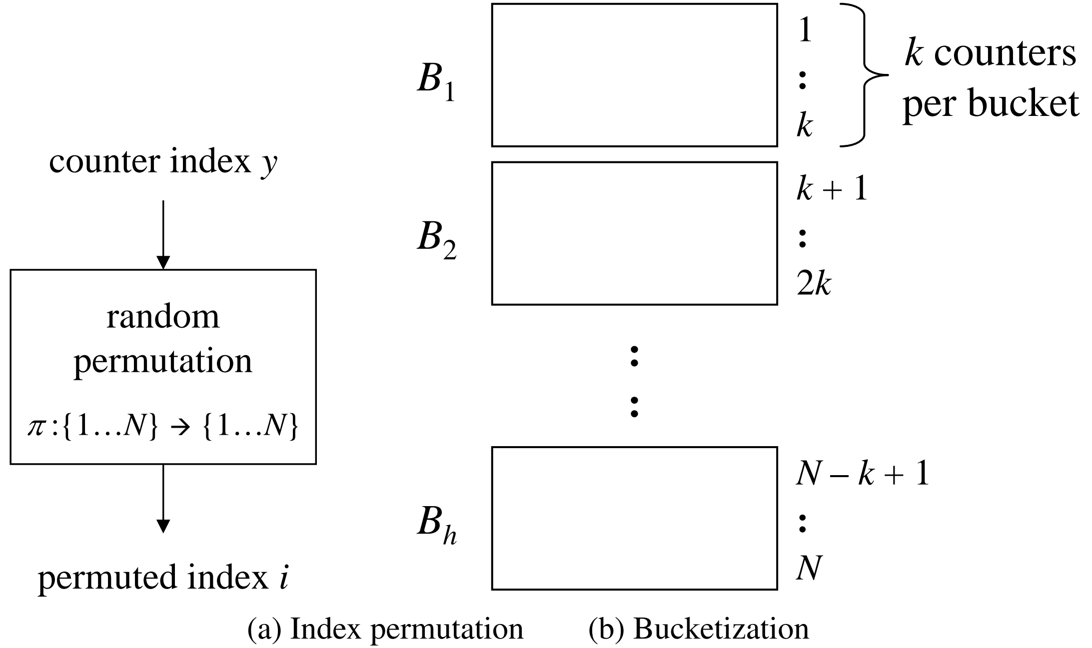
since only a tiny number of them will have counts large enough to require this worst-case width while most others need significantly fewer bits. Therefore, BRICK adopts a sophisticated *variable width encoding* of counters and can statistically multiplex these variable width counters through a bucketing scheme to achieve a much more compact representation. However, unlike the aforementioned baseline bucketing scheme, BRICK is extremely SRAM-efficient yet allows for very fast counter lookup and increment operations.

In the following, we will first present an overview of our proposed design in Section 3.3.1, followed by how it handles lookups, increments, and bucket overflows in Sections 3.3.2 to 3.3.4, respectively.

3.3.1 Overview

The basic idea of BRICK is to *randomly* bundle N counters into h buckets, B_1, B_2, ..., B_h, where each bucket holds k counters (e.g. $k = 64$ in practice) and $N = hk$. In each bucket, some counters will be long (possibly L bits in the worst-case) and some will be short, depending on the values they contain. As discussed earlier, the objective of bundling is to "statistically multiplex" the variable counter widths in a bucket so that each bucket only needs to be allocated memory space that is slightly larger than k times the average counter width (across N counters). Note that since we do not know the actual average width of a counter in advance, we need to instead use the *average width* in the following adversarial context. Imagine that an adversary chooses $C_1, C_2, \ldots C_N$ values under the constraint $\sum_{i=1}^{N} C_i \leq M$ that maximizes the metrics (e.g., average counter width). We emphasize that such an adversary is defined entirely in the well-established context of randomized online algorithm design [50] and has nothing to do with its connotation in security and cryptography.

Fig. 10 depicts these ideas of randomization and bucketization. In particular, as depicted in Fig. 10(a), to access the yth counter, a pseudorandom permutation function π :



(a) Index permutation (b) Bucketization

Figure 10: Randomly bundling counters into buckets.

$\{1\ldots N\} \rightarrow \{1\ldots N\}$ is first applied to the index y to obtain a permuted index i. This pseudo-random permutation function in practice can be as simple[3] as reversing the bits of y. The corresponding counter C_i can then be found in the ℓth bucket B_ℓ, where $\ell = \lceil \frac{i}{k} \rceil$. The bucket structure is depicted in Fig. 10(b). Unless otherwise noted, when we refer to the ith counter C_i, we will assume i is already the result of a random permutation.

As we explained before, the baseline bucketing scheme does not allow for efficient read and write (increment) accesses. In BRICK, a multi-level partitioning scheme is designed to address this problem as follows. The worst-case counter width L is divided into p parts, which we refer to as "sub-counters". The jth sub-counter, $j \in [1, p]$ (from the least significant bits to most significant bits) has w_j bits, such that $0 < w_j \leq L$ and $\sum_{j=1}^{p} w_j = L$. To save space, for each counter, BRICK maintains just enough of its sub-counters to hold its current value. In other words, counters with values no more than $2^{w_1+w_2+\cdots+w_i}$ will not have

[3]Since the adversary is defined in the online algorithm context discussed above, we do not believe cryptographically strong pseudorandom permutations, which may increase our cost and slow down our operations, are needed here.

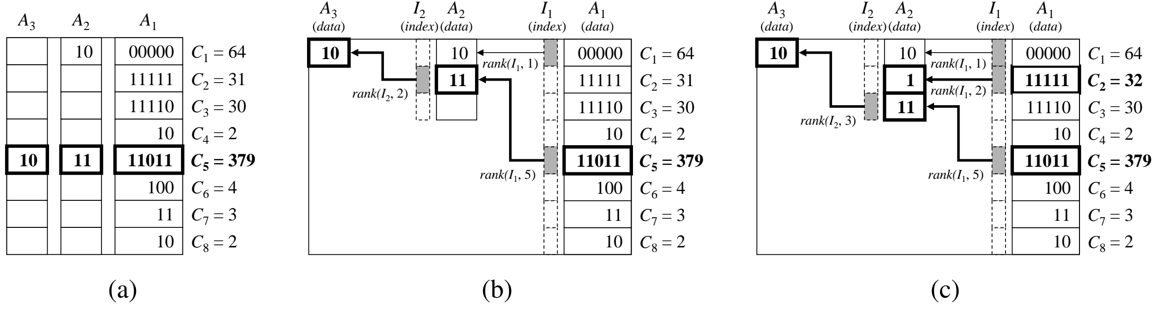


Figure 11: (a) Within a bucket, segmentation of variable-width counters into sub-counter arrays. (b) Compact representation of variable-width counters. (c) Updated data structure after incrementing C_2.

its $(i+1)^{th}, \ldots, p^{th}$ sub-counters stored in BRICK. For example, if $w_1 = 5$, any counter with value less than $2^5 = 32$ will only be allocated a memory entry for its 1st sub-counter. Consider the example shown in Fig. 11(a) with $k = 8$ counters in a bucket. Only C_1 and C_5 require more than their first sub-counters. Such an on-demand allocation requires us to link together all sub-counters of a counter, which we achieve using a simple and memory-efficient bitmap indexing scheme called *rank indexing*. Rank indexing enables efficient lookup as well as efficient expansion (when counter values exceed certain thresholds after increments), which will be discussed in detail in Section 3.3.2.

Each bucket contains p sub-counter arrays A_1, A_2, ..., A_p to store the 1st, 2nd, ..., p^{th} sub-counters (as needed) of all k counters in the bucket. How many entries should be allocated for each array A_i, denoted as k_i, turns out to be a non-trivial statistical optimization problem. On the one hand, to save memory, we would like to make $k_2, k_3, \ldots, k_p$ (k_1 is fixed as k) as small as possible. On the other hand, when we encounter the unlucky situation that we need to exceed any of these limits (say for a certain d, we have more than k_d counters in a bucket that have values larger than or equal to $2^{w_1+w_2+\cdots+w_{i-1}}$), then we will have a "bucket overflow" that would require that all counters inside the bucket be relocated to an additional array of full-size buckets with fixed worst-case width L for each counter, as we will show in Section 3.3.4. Given the high cost of storing a duplicated bucket in the full-size array, we would like to choose larger $k_2, \ldots, k_p$ to make this probability as small as possible.

3.3.2 Rank Indexing

A key technique in our data structure is an indexing scheme that allows us to efficiently identify the locations of the sub-counters across the different sub-counter arrays for some counter C_i. In particular, for C_i, its d sub-counters $C_{i,1}, \ldots, C_{i,d}$ are spread across $A_1, \ldots, A_d$ at locations $a_{i,1}, \ldots, a_{i,d}$, respectively (i.e., $C_{i,j} = A_j[a_{i,j}]$). For example, as shown in Fig. 11(b), C_5 is spread across $A_3[1] = 10$, $A_2[2] = 11$, and $A_1[5] = 11011$.

For each bucket, we maintain an index bitmap I. I is divided into $p-1$ parts, $I_1, \ldots, I_{p-1}$, with an one-to-one correspondence to the sub-counter arrays $A_1, \ldots, A_{p-1}$, respectively. Each part I_j is a bitmap with k_j bits, $I_j[1], \ldots, I_j[k_j]$, one bit $I_j[a]$ for each entry $A_j[a]$ in A_j. Each $I_j[a]$ is used to determine if the counter stored in $A_j[a]$ has expanded beyond the jth sub-counter array. I_j is also used to compute the index location of C_i in the next sub-counter array A_{j+1}. Because a counter cannot expand beyond the last sub-counter array, there is no need for an index bitmap component for the most significant sub-counter array A_p. For example, consider the entries $A_1[1]$ and $A_1[5]$ where the corresponding counter has expanded beyond A_1. This is indicated by having the corresponding bit positions $I_1[1]$ and $I_1[5]$ set to 1, as shown in shaded boxes in Fig. 11(b). All remaining bit positions in I_1 are set to 0, as shown in clear boxes.

For each counter that has expanded beyond A_1, an arrow is shown in Fig. 11(b) that links a sub-counter in A_1 with the corresponding sub-counter entry in A_2. For example, for C_5, its sub-counter entry $A_1[5]$ in A_1 is linked to the sub-counter entry $A_2[2]$ in A_2. Rather than expending memory to store these links explicitly, which could vanish savings gained by reduced counter widths, we *dynamically* compute the location of a sub-counter in the next sub-counter array A_{j+1} based on the current bitmap I_j. This way, no memory space is needed to store link pointers. This dynamic computation can be readily determined using an operation called `rank`(s, j), which returns the number of ones only in the range $s[1] \ldots s[j]$ in the bit-string s. This operation is similar to the `rank` operator defined in [35].

We apply the `rank` operator on a bitmap I_j by interpreting it as a bit-string. As we

shall see in Sections 3.4 and 3.6, our approach is designed to work with small buckets of counters (e.g. $k = 64$). Therefore, the corresponding bit-strings I_j are also relatively short since all sub-counter arrays satisfy $k_j \leq k$. Moreover, each successive k_j in the higher sub-counter arrays is substantially smaller than the previous sub-counter array, with the corresponding reduction in the length of the bit-string I_j. In turn, the `rank` operator can be efficiently implemented by combining a bitwise-AND instruction with another operation called `popcount`(s), which returns the number of ones in the bit-string s. Fortunately, the `popcount` operator is becoming an increasingly available hardware-optimized instruction in modern microprocessors and network processors. For example, current generations of 64-bit x86 processors have this instruction built-in [1, 2]. Using this `popcount` instruction, the `rank` operation for bit-strings with lengths up to $|s| = 64$ bits can be readily computed in as few as two instructions. As shown with numerical examples and trace simulations in Section 3.6, very good results can be achieved with a bucket size fixed at 64.

The pseudo-code for the `lookup` operation is shown in Algorithm 1. The retrieval of the sub-counters using rank indexing is shown in Lines 3-6, with the final count returned at the end of the procedure. For a hardware implementation, the iterative procedure can be readily pipelined. As we shall see in Section 3.6, we only need a small number of levels (e.g. three) in practice to achieve efficient results.

3.3.3 Handling Increments

The `increment` operation is also based on the traversal of sub-counters using rank indexing. We will first describe the basic idea by means of an example. Consider the counter C_2 in Fig. 11(b). Its count is 31, which can be encoded in just the sub-counter array A_1 with $C_{2,1} = 11111$. Suppose we want to increment C_2. We first increment its first sub-counter component $C_{2,1} = 11111$, which results in $C_{2,1} = 00000$ with a *carry propagation* to the next level. This is depicted in Fig. 11(c).

This carry propagation triggers the increment of the next sub-counter component $C_{2,2}$.

Algorithm 1: Pseudo-code

```
1  lookup(i)
2    C_i = 0; a = i  mod k;
3    for j = 1 to p
4      C_{i,j} = A_j[a];
5      if (j == p or I_j[a] == 0) break;
6      a = rank(I_j, a);
7    return C_i;

8  increment(i)
9    a = i  mod k;
10   for j = 1 to p
11     A_j[a] = A_j[a] + 1;
12     if (j == p or A_j[a] ≠ 0) break; /* last array or no carry */
13     if (I_j[a] == 1) /* next level already allocated */
14       a = rank(I_j, a);
15     else /* expand */
16       I_j[a] = 1;
17       a = rank(I_j, a);
18       b = (a − 1)w_{j+1} + 1;
19       A_{j+1} = varshift(A_{j+1}, b, w_{j+1});
20       I_{j+1} = varshift(I_{j+1}, a, 1);
21       A_{j+1}[a] = 1;
22       break;
```

The location of $C_{2,2}$ can be determined using rank indexing (i.e. $\texttt{rank}(I_1, 2) = 2$). However, the location of $A_2[2]$ was previously occupied by the counter C_5. To maintain *rank ordering*, we have to *shift* the entries in A_2 down by one to free up the location $A_2[2]$. This is achieved by applying an operation called $\texttt{varshift}(s, j, c)$, which performs a right shift on the sub-string starting at bit-position j by c bits (with vacant bits filled by zeros). The `varshift` operator can be readily implemented in most processors by means of shift and bitwise-logical instructions.

In particular, we can view a sub-counter array A_j as a *bit-string* formed by the concatenation of its entries, namely $A_j = A_j[1]A_j[2]\ldots A_j[k_j]$. The starting bit-position for an entry $A_j[a]$ in the bit-string can be computed as $b = (a-1)w_j + 1$, where w_j is the bit-width of the sub-counter array A_j. Consider C_5 in Fig. 11(c). After the shifting operation has been applied, the location of its sub-count in A_2 will be shifted down by one entry. Therefore,

its corresponding expansion status in I_2 must be shifted down by one position as well. The carry propagation of C_2 into A_2 is achieved by setting $A_2[2] = 1$.

As with the `rank` operator, BRICK has been designed to work with small fixed size buckets so that `varshift` can be directly implemented using hardware-optimized instructions. In particular, `varshift` only has to operate on A_2 or higher. Since the size of each level decreases exponentially, the bit-strings formed by each sub-counter array A_2 and above are also very short. As the results show in Section 3.6, with a bucket size of 64, all sub-counter arrays A_2 and above have a string length at most 64 bits, much less for the higher levels. Therefore, `varshift` can be directly implemented using 64 bit instructions.

The pseudo-code for the `increment` operation is shown in the latter part of Algorithm 1. Again, the iterative procedure shown in Algorithm 1 for `increment` is readily amenable to pipelining in hardware. In general, the lookup or update of each successive level of sub-counter arrays can be pipelined such that at each packet arrival, a lookup or update can operate on A_1 while a previous operation operates on A_2, and so forth.

3.3.4 Handling Overflows

Thus far, we have assumed in our basic data structure that we are guaranteed that each sub-counter array has been dimensioned to always provide sufficient entries to store all sub-counters in a bucket. To achieve greater memory efficiency, the number of entries in the sub-counter arrays can be reduced so that there is only a very small probability that a bucket will not have sufficient sub-counter array entries. As rigorously analyzed in Section 3.4 and numerically evaluated in Section 3.6, this bucket overflow probability can be made arbitrarily small while achieving significant reduction in storage for each bucket.

To facilitate this overflow handling, we extend the basic data structure described in Section 3.1 with a small number of *full-size* buckets $F_1, F_2, \ldots, F_J$. Each full-size bucket F_t is organized as k full-size counters (i.e., all counters with a worst-case width of L bits). When a bucket overflow occurs for some B_ℓ, the next available full-size bucket F_t is allocated to

store its k counters, where t is just +1 of the last allocated full-size bucket. An overflow status flag f_ℓ is set to indicate the bucket has overflowed. The index of the full-size bucket F_t is stored in a field labeled t_ℓ, which is associated with B_ℓ. In practice, we only need a small number of full-size buckets. As shown in Section 3.6, for real Internet traces with over a million counters, only about $J \approx 100$ full-size buckets are enough to handle the overflow cases. Therefore, the index field only requires a small number of extra bits per bucket (e.g. 7 bits).

Rather than migrating *all* k counters from B_ℓ to F_{t_ℓ} *at once*, a counter is only migrated *on-demand* upon the next increment operation ("migrate-on-write"). This way, the migration of an overflow counter to a full-size counter does not disrupt other counter updates. The location of counter C_i in F_{t_ℓ} is simply $a = i \mod k$, as before. To indicate if counter C_i has been migrated, a migration status flag $g_{t_\ell}[a]$ is associated with each counter entry $F_{t_\ell}[a]$ (i.e. $g_{t_\ell}[a] = 1$ indicates that the corresponding counter has been migrated).

The modified lookup operation simply first checks if a counter from an overflowed bucket has already been migrated, in which case the full-size count is simply retrieved from corresponding full-size bucket entry. Otherwise, the counter is retrieved as before. The modified increment operation is extended in a similar manner. It first checks if a counter from an overflowed bucket has already been migrated, in which case the full-size counter in the corresponding full-size bucket is incremented. If the counter is from a previously overflowed bucket B_ℓ, but it has not been migrated yet, then it is read from B_ℓ, incremented, and migrated-on-write to the corresponding location in the full-size bucket. Otherwise, the counter in B_ℓ is incremented as before. Finally, before propagating a carry to the next level, we first check if all entries in the next sub-counter array are already being used. If so, the next full-size bucket is allocated and the incremented count is migrated-on-write to the corresponding location.

3.4 Analysis

3.4.1 Analytical Guarantees

In this section, we bound the failure probability P_f that the number of overflowed buckets, each of which carries the hefty penalty of having to be allocated an additional bucket of full-size counters (as discussed in Section 3.3.4), will exceed any given threshold J. We will establish a rigorous relationship between P_f and parameters k_2, k_3, ..., k_p the number of entries BRICK allocates to sub-counter arrays $A_2, ..., A_p$ (The size of A_1 is already fixed to k) and w_1, w_2, ..., w_p, the widths of an entry in $A_2, ..., A_p$. The ultimate objective of this analysis is to find the optimal tradeoff between k_2, k_3, ..., k_p and J that allows us to minimize the amount of overall memory consumption ($h = N/k$ regular buckets + J full-size buckets) while keeping the failure probability P_f under an acceptable threshold (say 10^{-10} or even smaller). Surprisingly, the theory of stochastic ordering [51], which seems unrelated to the context of this work, plays a major role in these derivations.

Recall that the maximum counter width L is partitioned into sub-counter widths w_1, w_2, ..., w_p. Only counters whose value is larger than or equal to 2^{L_d}, where L_d is defined as $\sum_{j=1}^{d-1} w_j$, will need an entry in the sub-counter array A_d of a bucket. Since the aggregate count of all counters is no more than M, we know that there will be at most m_d of such counters in the whole counter array, where m_d is defined as $M2^{-L_d}$.

Now imagine at most m_d such counters are uniformly randomly distributed into N array locations through the aforementioned index permutation scheme. We hope that they are very evenly distributed among these buckets so that very few buckets will have more than k_d of them falling into it (i.e., overflow of A_d). Suppose we dimension J_d full-size buckets to handle bucket overflows caused by these counters. We would like to bound the probability that more than J_d buckets will have their A_d arrays overflowed.

We will consider the worst case scenario that there are exactly m_d counters needing entries in A_d. If there are less such counters, the overflow probability will only be smaller, and our tail bound still applies. For convenience, we denote the percentage of them in the

counter array $\frac{m_d}{N}$ as α_d.

Let random variables $X_{1,d}$, $X_{2,d}$, ..., $X_{h,d}$ be the number of used entries in the sub-counter array A_d among the buckets B_1, B_2, ..., B_h. Each array location has a probability α_d of being assigned one of the m_d counters, and there are k array locations in each bucket, so $X_{j,d}$ is roughly distributed as $Binomial(k, \alpha_d)$ for any j. Here $Binomial(\mathcal{N}, \mathcal{P})$ is the Binomial distribution with $\mathcal{N}$ trials and $\mathcal{P}$ as the success probability of each trial. Therefore, the overflow probability of level d from any bucket B_j is roughly

$$\epsilon_d = \mathcal{B}inotail_{k,\alpha_d}(k_d)$$

where $\mathcal{B}inotail_{\mathcal{N},\mathcal{P}}(\mathcal{K}) \equiv \sum_{z=\mathcal{K}+1}^{\mathcal{N}} \binom{\mathcal{N}}{z} \mathcal{P}^z (1-\mathcal{P})^{(\mathcal{N}-z)}$ denotes the tail probability $Pr[Z > \mathcal{K}]$, where Z has distribution $Binomial(\mathcal{N}, \mathcal{P})$.

Intuitively, these random variables are *almost independent*, as the only dependence among them seems to be that their total is m_d. If we do assume that they are independent, then the probability that the number of total overflows be larger than J_d entries is roughly

$$\delta_d = \mathcal{B}inotail_{h,\epsilon_d}(J_d)$$

Readers understandably will immediately protest this voodoo tail bound result since the $X_{j,d}$'s are not exactly Binomial, and they are not actually independent. Interestingly, we are able to establish a rigorous tail bound of $2\delta_d$, which is only two times the voodoo tail bound δ_d. A similar bound has been established by Mitzenmacher and Upfal in their book [48] which used independent Poisson distributions to bound multinomial distributions, using techniques from stochastic ordering theory [51] implicitly (i.e., without introducing such concepts). In our case we use independent binomial distributions to bound multivariate hypergeometric distributions, i.e. those of $X_{1,d}$, $X_{2,d}$, ..., $X_{h,d}$.

Based on this rigorous tail bound to be proven in Section 3.4.2 and taking union of the overflow events from all the subarrays, we arrive at the following corollary.

Corollary 1. *Let parameters δ_2, $\cdots$, δ_p be defined as above. The failure probability of*

insufficient full-size buckets, i.e. that the total number of overflows that need to be moved to the additional full-size buckets from all subarrays exceeds $J = J_2 + \cdots + J_p$, is no more than $2(\delta_2 + \cdots + \delta_p)$.

If given a target worst-case failure probability P_f of insufficient full-size buckets, e.g. 10^{-10} or even smaller, an optimization procedure remains to configure parameters from w_1 to w_{p-1}, k_2 to k_p, and J_2 to J_p, so that we can achieve the best tradeoff for the overall memory space, which takes into consideration the storage of all sub-counter arrays, index bitmaps, and all full-size buckets, and even the $\lfloor \lg(J) \rfloor + 2$ bits for f_ℓ and t_ℓ in each bucket, which indicate the migration to full-size buckets.

Given the messy nature of the Binomial distribution, "clean" analytical solutions (e.g., based on Lagrange multipliers) do not exist. We designed a quick search strategy that can generate near-optimal configurations. Our evaluation results in Sec 3.6 are obtained based on the near-optimal parameter configurations generated by this procedure. We omit the detail of this procedure in the interest of space.

3.4.2 the Main Tail Bound

In this section, we state formally the aforementioned tail bound theorem (two times the voodoo bound). We would like to state this theorem using generic parameters that have the same symbol as before but without the subscript d, since they can be replaced by the corresponding parameters with subscript d to obtain the tail bound on the number of overflows from every subarray A_d. In particular, we will replace m_d (the number of counters that will have an entry in sub-counter array A_d) by m, and k_d (the number of entries in sub-counter array A_d) by c, as k has been used to denote the number of counters in each bucket in the original counter array. Furthermore, to highlight the general nature of our theorem, we further detach ourselves from the application semantics by stating the theorem as follows.

Theorem 2. *m balls are uniformly randomly thrown into h buckets that has k entries each, with at most one ball in each entry. Let $N = hk$. Let $X_1^{(m)}$, ..., $X_h^{(m)}$ be the number of balls*

that falls into each bucket. Let $\alpha = \frac{m}{hk}$ and assume $\alpha \leq \frac{1}{2}$. Let $Y_1^{(\alpha)}$, ..., $Y_h^{(\alpha)}$ be independent random variables distributed as Binomial(k, α). Let $f(x_1, ..., x_h)$ be an increasing function in each argument. Then

$$E[f(X_1^{(m)}, ..., X_h^{(m)})] \leq 2E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)})]$$

Before we prove this theorem, we need to formally characterize the underlying probability model and in particular specify precisely what we mean by throwing m balls "uniformly randomly" into N entries as follows. Among all $\binom{N}{m}$ ways of injective mapping from m balls into N entries, every way happens with equal probability $\frac{1}{\binom{N}{m}}$, when these balls are considered indistinguishable. We refer to this characterization of the underlying probability model as "throwing m balls into N entries in one shot". It is not hard to verify that the following process of "throwing m balls into N entries one by one" results in the same probability model. In this process, at first a ball is thrown into an entry chosen uniformly from these N entries. Then another ball is thrown into an entry uniformly picked from the remaining $N-1$ entries, and so on. This equivalent characterization of the underlying probability model makes it easier for us to establish the stochastic ordering relationship among vectors of random variables in Section 3.4.3, an essential step for the proof of Theorem 2. Now we are ready to prove Theorem 1.

Proof of Theorem 1. We use $X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}$ when there are l balls thrown instead of m. In Proposition 3, we prove that for any l value, $\mu(X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)})$ is equivalent to $\mu(Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)} | \sum_{j=1}^{h} Y_j^{(\alpha)} = l)$, where $\mu(Z)$ denotes the distribution of a random variable or vector Z. In other words, conditioned upon $\sum_{j=1}^{h} Y_j^{(\alpha)} = l$, the independent random variables $Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)}$ have the same joint distribution as dependent random variables $X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}$. Then we prove in Proposition 5 that, when $l \leq l'$, $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$ is stochastically less than or equal to (defined later) $[X_1^{(l')}, X_2^{(l')}, \ldots, X_h^{(l')}]$. For any increasing function $f(x_1, x_2, ..., x_h)$, we have

$$
\begin{aligned}
& E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)})] \\
= \ & \sum_{l=0}^{N} E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)}) \mid \sum_{j=1}^{h} Y_j^{(\alpha)} = l] \Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} = l] \\
\geq \ & \sum_{l=m}^{N} E[f(Y_1^{(\alpha)}, ..., Y_h^{(\alpha)}) \mid \sum_{j=1}^{h} Y_j^{(\alpha)} = l] \Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} = l] \\
= \ & \sum_{l=m}^{N} E[f(X_1^{(l)}, ..., X_h^{(l)})] \Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} = l] \qquad (3) \\
\geq \ & \sum_{l=m}^{N} E[f(X_1^{(m)}, ..., X_h^{(m)})] \Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} = l] \qquad (4) \\
= \ & E[f(X_1^{(m)}, \cdots, X_h^{(m)})] \Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} \geq m] \\
= \ & E[f(X_1^{(m)}, \cdots, X_h^{(m)})] \mathcal{B}inotail_{N,\alpha}(m-1) \\
\geq \ & \frac{1}{2} E[f(X_1^{(m)}, ..., X_h^{(m)})] \qquad (5)
\end{aligned}
$$

Equality (3) is due to Proposition 3, inequality (4) is due to Proposition 3, and inequality (5) is due to the properties of the 50-percentile point of Binomial distributions proven in [27]. □

Corollary 2. *Let the variable be as defined in Theorem 2. Let c and J be some constants. Let $\epsilon = \mathcal{B}inotail_{k,\alpha}(c)$. Then*

$$
Pr[\sum_{j=1}^{h} 1_{\{X_j^{(m)} > c\}} > J] \leq 2\mathcal{B}inotail_{h,\epsilon}(J)
$$

Proof. Consider function $f(x_1, x_2, \cdots, x_h) \equiv 1_{\{\sum_{j=1}^{h} 1_{\{x_j > c\}} > J\}}$, which is an increasing function of x_1, ..., x_h. From Theorem 2 we have $\Pr[\sum_{j=1}^{h} 1_{\{X_j^{(m)} > c\}} > J] \leq 2 \Pr[\sum_{j=1}^{h} 1_{\{Y_j^{(\alpha)} > C\}} > J]$. Since $\{1_{\{Y_j^{(\alpha)} > c\}}\}_{1 \leq j \leq h}$ are independent Bernoulli random variables with probability $\epsilon = \mathcal{B}inotail_{k,\alpha}(c)$, their sum is distributed as $Binomial(h, \epsilon)$. Therefore $\Pr[\sum_{j=1}^{h} 1_{\{Y_j > c\}} > J]$ is equal to $\mathcal{B}inotail_{h,\epsilon}(J)$. □

3.4.3 Proofs of propositions 1–3

Proposition 3.

$$\mu(X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}) = \mu(Y_1^{(\alpha)}, Y_2^{(\alpha)}, \ldots, Y_h^{(\alpha)} | \sum_{j=1}^{h} Y_j^{(\alpha)} = l) \tag{6}$$

Proof. It suffices to prove that for any nonnegative integers $l_1, l_2, ..., l_h$ that satisfy $\sum_{j=1}^{h} l_j = l$, $\Pr[X_1^{(l)} = l_1, X_2^{(l)} = l_2, \ldots, X_h^{(l)} = l_h] = \Pr[Y_1^{(\alpha)} = l_1, Y_2^{(\alpha)} = l_2, \ldots, Y_h^{(\alpha)} = l_h \mid \sum_{j=1}^{h} Y_j^{(\alpha)} = l]$. We show that both the LHS (left hand side) and the RHS (right hand side) are equal to

$$\frac{\binom{k}{l_1}\binom{k}{l_2}\cdots\binom{k}{l_h}}{\binom{N}{l}} \tag{7}$$

Since there are $\binom{N}{l}$ ways of selecting l entries out of a total of N entries, and each way happens with equal probability $\frac{1}{\binom{N}{l}}$, the LHS is equal to (7) because there are $\binom{k}{l_1}\binom{k}{l_2}\cdots\binom{k}{l_h}$ ways among them that result in the event $\{X_1^{(l)} = l_1, X_2^{(l)} = l_2, \ldots, X_h^{(l)=l_h}\}$. Now we prove that the RHS is equal to (7) as well. Since $Y_1^{(\alpha)}$, $Y_2^{(\alpha)}$, ..., $Y_h^{(\alpha)}$ are independent random variables with distribution $Binomial(k, \alpha)$, $\sum_{j=1}^{h} Y_j^{(\alpha)}$ has distribution $Binomial(N, \alpha)$ and therefore

$$\Pr[\sum_{j=1}^{h} Y_j^{(\alpha)} = l] = \binom{N}{l}\alpha^l(1-\alpha)^{N-l} \tag{8}$$

Additionally, when $\sum_{j=1}^{h} Y_j^{(\alpha)} = l$, we have

$$\begin{aligned} & \Pr[Y_1^{(\alpha)} = l_1, Y_2^{(\alpha)} = l_2, \ldots, Y_h^{(\alpha)} = l_h, \sum_{j=1}^{h} Y_j^{(\alpha)} = l] \\ = & \prod_{j=1}^{h}\binom{k}{l_j}\alpha^{l_j}(1-\alpha)^{k-l_j} = \alpha^l(1-\alpha)^{N-l}\prod_{j=1}^{h}\binom{k}{l_j} \end{aligned} \tag{9}$$

Combining (8) and (9) we obtain that the RHS is equal to (7) as well. □

Stochastic ordering is a way to compare two random variables. Random variable X is stochastically less than or equal to random variable Y, written $X \leq_{st} Y$, iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. An equivalent definition of $X \leq_{st} Y$ is that $Pr[X > t] \leq Pr[Y > t], -\infty < t < \infty$. The definition involving increasing

functions also applies to random vectors $X = (X_1, ..., X_h)$ and $Y = (Y_1, ..., Y_h)$: $X \leq_{st} Y$ iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exits. Here ϕ is increasing means that it is increasing in each argument separately with other arguments being fixed. This is equivalent to $\phi(X) \leq_{st} \phi(Y)$. Note this definition is a much stronger condition than $Pr[X_1 > t_1, ..., X_h > t_h] \leq Pr[Y_1 > t_1, ..., Y_h > t_h]$ for all $t = (t_1, ..., t_h) \in \mathcal{R}^n$.

Now we state without proof a fact that will be used to prove Proposition 5. Its proof can be found in all books that deal with stochastic ordering [51].

Proposition 4. *Let X and Y be two random variables (or vectors). $X \leq_{st} Y$ iff there exists X' and Y' such that $\mu(X') = \mu(X)$, $\mu(Y') = \mu(Y)$, and $Pr[X' \leq Y'] = 1$.*

Now we are ready to prove the following proposition.

Proposition 5. *For any $0 \leq l < l' \leq N$, we have*

$$[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}] \leq_{st} [X_1^{(l')}, X_2^{(l')}, \ldots, X_h^{(l')}] \tag{10}$$

Proof. It suffices to prove it for $l' = l + 1$. Our idea is to find random variables Z and W such that Z has the same distribution as $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$, W has the same distribution as $[X_1^{(l+1)}, X_2^{(l+1)}, \ldots, X_h^{(l+1)}]$, and $\Pr[Z \leq W] = 1$. We will use the aforementioned probability model that is generated by "throwing m balls into N entries one-by-one" random process. Now given any outcome ω in the probability space Ω, let $Z(\omega) = [Z_1(\omega), Z_2(\omega), ..., Z_h(\omega)]$, where $Z_j(\omega)$ is the number of balls in the j_{th} bucket after we throw l balls into these N entries one by one. Now with all these l balls there, we throw the $(l+1)_{th}$ ball uniformly randomly into one of the remaining empty entries. We define $W(\omega)$ as $[W_1(\omega), W_2(\omega), ..., W_h(\omega)]$, where $W_j(\omega)$ is the number of balls in the j_{th} bucket after we throw in the $(l + 1)_{th}$ ball. Clearly we have $Z(\omega) \leq W(\omega)$ for any $\omega \in \Omega$ and therefore $\Pr[Z \leq W] = 1$. Finally, we know from the property of the "throwing m balls into N entries one-by-one" random process that Z and W have the same distribution as $[X_1^{(l)}, X_2^{(l)}, \ldots, X_h^{(l)}]$ and $[X_1^{(l+1)}, X_2^{(l+1)}, \ldots, X_h^{(l+1)}]$ respectively. □

3.5 *Information theory bound*

We are interested in how far we are from the optimal memory usage cost. In this section, we will try to answer this question partly by deriving several lower bounds on the minimum memory requirement per counter under the aforementioned constraint that the sum of counter values C_1, C_2, ..., C_N (i.e., the total number of increments) is no more than M.

We will explore this question in the following sequence. Firstly, we will present a naive bound, which is the worst-case total number of bits needed to store all these counter values. However, this bound doesn't take into account the indexing cost. Therefore, secondly, we will explore the additional indexing cost based on information theory. However, this part is only aimed at getting a feeling about how much additional indexing cost is required, since the derivation is not strict. Finally, we will analyze and prove the minimum number of bits needed to accurately represent the whole counter array, no matter what kind of coding/decoding techniques are used.

3.5.1 The worst-case average binary length of counters

When no coding techniques are used, every counter value is stored as a plain binary number. The minimum memory required are the worst-case total number of bits of those counter values.

For each counter value C_i, its binary length is $\lfloor \log_2 C_i \rfloor + 1$, where we use the convention $\log_2(0) = 0$. Hence the worst-case bound of the average[4] number of bits per counter, defined as B_0, is as follows:

$$B_0 = \max_{\sum_{i=1}^{N} C_i \leq M} \sum_{i=1}^{N} \frac{\lfloor \log_2 C_i \rfloor + 1}{N} = \max_{\sum_{i=1}^{N} C_i = M} \sum_{i=1}^{N} \frac{\lfloor \log_2 C_i \rfloor + 1}{N}.$$

It could be bounded through the Jensen's inequality:

[4]For the convenience of comparison, we calculate and compare the **average** number of bits per counter instead of the total number.

$$B_0 \leq \sum_{i=1}^{N} \frac{\log_2 C_i + 1}{N} = \sum_{i=1}^{N} \frac{\log_2 C_i}{N} + 1$$
$$\leq \log_2(\frac{1}{N}\sum_{i=1}^{N} C_i) + 1 = \log_2(M/N) + 1$$

If $\log_2 \frac{M}{N}$ is an integer, all the inequalities above would hold when all counter values are equal to $\frac{M}{N}$, and the bound B_0 would be reached at $\log_2 \frac{M}{N} + 1$.

When $\log_2 \frac{M}{N}$ is not an integer, it becomes a little more complex to get the precise value of B_0. We claim B_0 would be bounded by the result of the following optimization:

$$B_0 = \max \sum_{i=1}^{L} i\beta_i$$

$$\textit{subject to} \qquad \forall i, \beta_i \geq 0; \sum_{i=1}^{L} \beta_i = 1 \tag{11}$$

$$M/N + 1 \leq \sum_{i=1}^{L} \beta_i 2^i \leq 2M/N \tag{12}$$

where β_i is the fraction of i-bit-long counters among all counters, i.e. $\frac{1}{N} \sum_{j=1}^{N} 1_{\lfloor \log_2 C_j \rfloor = i-1}$, L is the maximum possible length of one counter, i.e. $\lfloor \log_2 M \rfloor + 1$. The constraints (12) are transformed from the constraint $\sum_{i=1}^{N} C_i = M$, since for each C_i that is j-bit-long, $2^{j-1} \leq C_i \leq 2^j - 1$.

We could numerically solve the linear programming above, and could find that the bound B_0 is at least $\log_2 \frac{M}{N} + 0.9$ for arbitrary $\frac{M}{N}$.

3.5.2 Bound$_1$: Considering the unavoidable indexing costs

The lower bound above does not account for the extra bits needed to delimit these counter values. However, it is hard to directly get the lower bound of indexing cost since we would never know whether we have invented the most efficient indexing scheme and how far it is from the optimal cost. We can only approximately estimate the cost through information entropy.

Given an array of indistinguishable counter bits, the original counter values could be decoded if and only if the sequence of the sizes of the counters are known. Hence we could

Table 9: Bounds for schemes without using any coding technique

$\log_2(\frac{M}{N})=$		2	4	6	8
B_0	$=\log_2 \frac{M}{N}+$	1.00	1.00	1.00	1.00
B_{index}	$=$	1.83	2.61	3.10	3.45
B_1	$=\log_2 \frac{M}{N}+$	1.82	1.94	1.96	1.97

model the worst-case indexing cost by calculating the worst-case information entropy of the sequence of the counter sizes under all possible distributions as follows:

$$B_{index} = \max_{\text{subject to (11),(12)}} \left\{ -\sum_{i=1}^{L} \beta_i \log_2 \beta_i \right\}$$

However, we should notice that the counter size distribution $\{\beta_1, \beta_2, \ldots\}$ for the worst-case indexing cost may not be the same as the distribution for the worst-case counter bits. Hence if we want a more reasonable bound, it should be the result of the following optimization of the sum of both the indexing entropy and the counter bits:

$$B_1 = \max_{\text{subject to (11),(12)}} \left\{ -\sum_{i=1}^{L} \beta_i \log_2 \beta_i + \sum_{i=1}^{L} i\beta_i \right\}$$

Both two bounds could solved by standard Lagrange techniques. The numerical result of B_0, B_{index} and B_1 are presented in Table 9. Although the worst-case indexing cost B_{index} could be very large, only bound B_0 and B_1 are comparable with our scheme. Compared with numbers in Table 11, our schemes are about 3 to 4 bits from the optimal cost that one could achieve without compressing the original counter bits.

3.5.3 Bound$_2$: Lower bound when optimal coding is used

Although we haven't seen any works that could use coding techniques and support router-level fast random read and write at the same time, we are still interested in the optimal memory cost if we allow coding techniques. In our previous publication [32] published in ANCS'08 , we calculate the worst-case 2-D empirical information entropy of the whole counter array and claim that it is the worst-case bound for the case when optimal coding is used. However, the method employed by [32] suffers the similar weakness as B_1 and hence not strict.

Table 10: Information-theoretic lower bound.

$\log_2(\frac{M}{N})$=	2	4	6	8
$B_2 \quad =\log_2 \frac{M}{N}+$	1.61	1.49	1.45	1.45

We propose the following method to calculate a strict worst-case bound. Our constraint $\sum_{i=1}^{N} C_i \leq M$ is equivalent to $\sum_{i=1}^{N} C_i + \delta = M$, where δ is a non-negative integer. Basic combinatorics gives that the number of distinct non-negative integer vectors $\{C_1, ..., C_N, \delta\}$ satisfying $\sum_{i=1}^{N} C_i + \delta = M$ is exactly $\binom{M+N}{N}$. Since each possible counter value vector must correspond to a distinct memory state, at least $\log_2 \binom{M+N}{N}$ bits of memory are needed to accurately represent all possible counter value vectors, no matter what coding technique is employed. So we get the lower bound on average number of bits per counter as $B_2 = \log_2 \binom{M+N}{N}/N$. This lower bound is also achievable in theory, since we can index all possible counter value vectors from 1 through $\binom{M+N}{N}$ and simply store the index as a binary number.

Using Sterling's Formula we could get,

$$B_2 = \log_2 \binom{M+N}{N}/N \approx \log_2 \left(\frac{M}{N}(1+\frac{N}{M})^{\frac{M}{N}+1}\right) \approx (\log_2 \frac{M}{N} + 1.44)$$

The numerical values of B_2 is presented in Table 10. Interestingly, the results are very close to the worst-case empirical entropy results in [32], with differences less than 1×10^{-4}.

We observe that the worst-case bound B_2, which allows coding techniques to be used on counter values, would be smaller than B_1 by around 0.2 to 0.5 bits. Hence we could conclude that we are only 4 to 6 bits away from the optimal cost we could get even if optimal coding is used.

We should notice that in reality we will never be even close to the bounds above, even the bound B_1, because the information theory employed in all those calculations does not take account of the complexity issues, considering our application scenarios requires encoding and decoding (write and read) to be both very fast.

3.6 Performance Evaluations

In this section, we will evaluate the performance of BRICK and show that BRICK is extremely memory-efficient. Our results show that the number of extra bits needed per counter in addition to the lower bounds $\log_2 \frac{M}{N}$ remains practically constant with increasing number of flows N, and hence the solution is scalable. We also evaluate in Section 3.6.2 the performance of our architecture using two real-world Internet traffic traces. Finally, we discuss implementation issues in Section 3.7.

3.6.1 Numerical Results of Analytical Bounds

In this section, we present a set of numerical results computed from the tail bound theorems derived in Section 3.4.

3.6.1.1 Configuration of Parameters and Memory Costs Optimization

Recall that in our problem, the number of flows N and the maximum total increments in a measurement period M are given. For the specified N and M, we apply our tail bound theorems to derive optimal configurations for different combinations of constraints, i.e., bucket sizes k, number of levels p, and failure probabilities P_f. The derived configuration is in terms of the number of entries in each sub-counter array k_j, the width of each sub-counter array w_j, and the number of full-size buckets J that we need to ensure a failure probability less than P_f (the probability that we have insufficient entries in a sub-counter array or a full-size bucket).

For a configuration of these parameters, the amount of memory required, which is also the target of our optimization, can be computed as follows:

$$\mathcal{S}_\ell = \left(\left[\sum_{j=1}^{p} k_j (w_j + 1) \right] - k_p \right) + (\lfloor \log_2 J \rfloor + 2) \tag{13}$$

$$\mathcal{S} = \left(\sum_{\ell=1}^{h} \mathcal{S}_\ell \right) + Jk\,(L + 1) \tag{14}$$

Here $\mathcal{S}_\ell$ is the memory cost of each bucket; its first component corresponds to the space

required for storing the sub-counter arrays and the index bitmaps, and its second component corresponds to the overflow status flag and the index to the corresponding full-size bucket[5]. Then the total memory cost $\mathcal{S}$ is $h = \lceil \frac{N}{k} \rceil$ buckets of size $\mathcal{S}_\ell$ each plus J full-size buckets of size $k(L+1)$ each. (For each full-size counter of size L, we need 1 bit for indicating the migration status.)

We traverse typical possible configurations with the help of branch-and-prune technique and select the best among those configurations. We should emphasize that, actually there might exist many different configurations around the optimal point. The difference between the average memory utilization of those quasi-optimal points is very minute. In practice, it is not necessary to really find the "optimal" point as long as the solution found is close to the optimal enough. Hence the results shown below are only of typical example and might not be the best.

3.6.1.2 Numerical results with various configurations

In Section 3.5, we have noticed that all lower bounds are just around one or two bits over $\log_2 \frac{M}{N}$. Hence, for convenience, we use $\sigma = (\mathcal{S}/N) - \log_2(M/N)$as a metric to evaluate the space efficiency of our solution.

In Table 11, we first consider results for the case with $k = 64$ counters per bucket. We first consider this case with a small bucket size to ensure that all string operations are within 64 bits, which allows for direct implementations using 64-bit instructions in modern processors [1, 2]. As we shall see, substantial statistical multiplexing can already be achieved with $k = 64$. For the results presented in Table 11, we used representative parameters with $N = 1$ million counters and $M = 16$ million as the maximum total increments during a measurement period. We also set the failure probability to be $P_f = 10^{-10}$, which is a tiny probability corresponding to an average of one failure (when there are more than J *overflowed buckets*) every ten thousand years. We will later show in Figures 13, 14, and 15 that

[5]Since there are J full-size buckets, this index can be stored in $\lfloor \log_2 J \rfloor + 1$ bits.

Table 11: An Example of Sub-counter array sizing and per-counter storage for $k = 64$ and $P_f = 10^{-10}$.

(a) Sizing of sub-counter arrays.

p	k_2	k_3	k_4	k_5	w_1	w_2	w_3	w_4	w_5
3	15	3			$\log_2 \frac{M}{N} + 3$	4	13		
4	25	10	2		$\log_2 \frac{M}{N} + 2$	2	4	12	
5	25	10	3	1	$\log_2 \frac{M}{N} + 2$	2	3	4	9

(b) Size of each sub-counter array = $k_j \times w_j$ (in bits).

p	A_2	A_3	A_4	A_5
3	$15 \times 4 = 60$	$3 \times 13 = 39$		
4	$25 \times 2 = 50$	$10 \times 4 = 40$	$2 \times 12 = 24$	
5	$25 \times 2 = 50$	$10 \times 3 = 30$	$3 \times 4 = 12$	$1 \times 9 = 9$

(c) Storage per counter.

$p = 3$	$p = 4$	$p = 5$
$\log_2 \frac{M}{N} + 6.05$	$\log_2 \frac{M}{N} + 5.66$	$\log_2 \frac{M}{N} + 5.50$

the additional per-counter storage cost beyond the minimum width of the average count is practically a constant unrelated to the number of flows N, the maximum total increments M, or the failure probability P_f.

In Table 11(a), the number of entries and the width for each sub-counter array are shown for BRICK implementations with varying number of levels p. As can be seen, in each design, the number of entries decreases exponentially as we go to the higher sub-counter arrays. This is the main source of our compression. With $k = 64$, the rank indexing operation described in Section 3.3 only needs to be performed on bitmaps with $|I_j| \leq 64$ bits (much less than 64 for the higher sub-counter arrays) and can be directly implemented using 64-bit popcount and bitwise-logical instructions that are available in modern processors [1, 2]. Table 11(b) shows the size of each sub-counter array. For all three designs, the space requirement for each sub-counter array other than A_1 is also less than 64 bits. Therefore, the "varshift" operator described in Section 3.3.3, which only needs to operate on A_2 and higher, can be directly implemented using 64-bit shift and bitwise-logical instructions as well.

In Table 11(c), the per-counter storage cost for the three designs are shown. For three

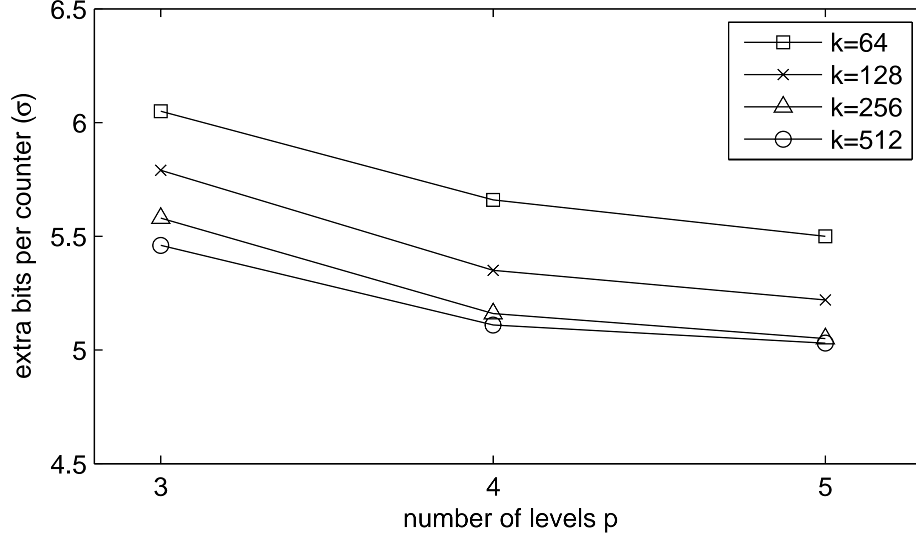


Figure 12: Impact of increasing bucket size k. Extra bits σ in the range of $[5.03, 6.05]$.

levels, the extra storage cost per counter is 6.05, and the extra storage costs per counter are 5.66 and 5.50 for four and five levels, respectively. The amount of extra storage only decreases slightly with additional levels in the BRICK implementation. For example, as we go from three to five levels, the reduction of $6.05 - 5.50 = 0.55$ extra bits is only about 5.5% in the overall per-counter cost if $\log_2 \frac{M}{N} = 4$.

We next consider the impact of larger bucket sizes on storage costs. Figure 12 shows the results for $k = 128$, 256, and 512. The number of extra bits per counter decreases with increasing bucket sizes and number of levels, with σ in the range of $[5.03, 6.05]$. The results show that increasing the bucket size has only an insignificant impact on the storage savings, corresponding to only a small increase in statistical multiplexing with larger buckets. Therefore, we will use 64 counters per bucket for software implementation and recommend it for ASIC implementation as well, since it has the advantage that operations can be directly implemented using 64-bit processor instructions in software.

As stated earlier, the added per-counter cost is practically a constant with respect to the number of flows N, the maximum total increments M, and the failure probability P_f. We now verify this statement by evaluating BRICK under different N, M, and P_f. The results

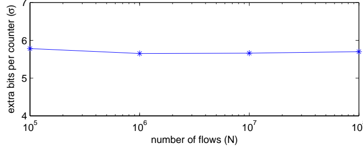


Figure 13: Impact of increasing number of flows N. Extra bits σ in the range of $[5.65, 5.78]$.

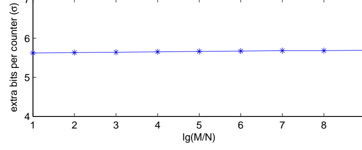


Figure 14: Impact of increasing $\log_2 \frac{M}{N}$. Extra bits σ in the range of $[5.62, 5.69]$.

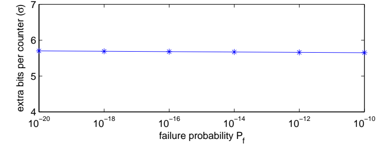


Figure 15: Impact of decreasing failure probability P_f. Extra bits σ in the range of $[5.65, 5.70]$.

are shown in Figures 13, 14, and 15, respectively. In these evaluations, we used 64 counters per bucket and four levels.

In Figures 13, 14, and 15 we evaluate the impact of different N, M, and P_f, where we use $k = 64$ and $p = 4$.

Figure 13 shows that the added per-counter cost remains practically constant as we increase N exponentially by powers of 10. Similarly, Figure 14 shows that the added cost also remains practically constant with different ratios of M and N. These results show that BRICK is scalable to different values of M and N with per-counter storage cost within approximately a constant factor from the minimum width of the average count. Figure 15 shows the impact of decreasing failure probability. We show results for $P_f = 10^{-10}$ down to 10^{-20}. Again, we see that the change in storage cost is negligible with decreasing failure probability, which means BRICK can be optimized to vanishingly small failure probabilities with virtually no impact on storage cost.

3.6.2 Results for real Internet traces

In this section, we evaluate our active counter architecture using real-world Internet traffic traces. The traces that we used were collected at different locations in the Internet, namely University of Southern California (USC) and University of North Carolina (UNC), respectively. The trace from USC was collected at their Los Nettos tracing facility on February 2, 2004, and the trace from UNC was collected on a 1 Gbps access link connecting the campus to the rest of the Internet on April 24, 2003. For each trace, we used a 10-minute

segment, corresponding to a measurement epoch. The trace segment from USC has 18.9 million packets and around 1.1 million flows; the trace segment from UNC has 32.6 million packets and around 1.24 million flows. We use the first counter for the first encountered flow in the trace, the second counter for the second encountered flow in the trace, etc.

We use the same general parameter settings as the evaluations in Section 3.5 with 64 counters per bucket, four levels, and a failure probability of $P_f = 10^{-10}$. The total storage space required for counting packets [6] in USC trace is 1.36 MB, and the total required for the UNC trace is 1.61 MB. In comparison, a naive implementation would require a worst-case counter width for all counters. Both traces require a worst-case width of 25 bits, whereas the BRICK implementations require a per-counter cost of about 10 bits. The total storage required for a naive implementation is 3.85 MB for the USC trace and 4.40 MB for the UNC trace. The BRICK implementations represent a 2.5x improvement in both cases. This is exciting since with the same amount of memory, we will be able to squeeze in 2.5 times more counters, which is badly needed in future faster and "more crowded" Internet!

Table 12: Percentage of full-size buckets.

Trace	h	J	$\frac{J}{h}$	J^*	J^{**}
USC	17.3K	112	0.65%	99	0
UNC	19.5K	127	0.65%	172	0

Table 12 shows the number of full-size buckets needed according to our tail bounds, and the number of full-size buckets actually used. We should emphasize that the numbers shown in Table 12 are only one example of the various good configurations selected by the process described in Section 3.6.1.1. The total number of full-size buckets needed could be much smaller by two to ten folds, at the cost of a little more total memory utilization (typically around 1 to 5%).

[6] We note that the memory savings for counting bytes would be less, due to the much larger M. Considering the typical average packet size is around 500 bytes, 9 more bits are needed per counter for both the naive implementation and the BRICK implementation.

In Table 12, we see that only a small number of full-size buckets are needed to guarantee a tiny probability ($P_f = 10^{-10}$) that we will have insufficient number of full-size buckets to handle bucket overflows. J^* denotes the actual number of full-size buckets used when no random permutation is used. J^{**} denotes the actual number of full-size buckets used when permutation is done by reversing bits. We could see that J^* is similar to or even worse than the calculated J. This is because that our guarantee is based on randomized permutation, and for J^* no permutation is used. In this experiment, the larger flows tend be concentrated in lower index counters, thus causing those buckets to overflow to full-size buckets. However, when some very simple "randomization" techniques are used, such as reversing the bits of the counter index, we could see that actually no full-size buckets are used for both traces. We emphasize that the J full-size buckets are allocated for guaranteeing a tiny probability ($P_f = 10^{-10}$) of overflow for any counter value distribution. In summary, this experiment demonstrates random permutations is crucial to our design and to establish of our statistical guarantee, and verifies that simple permutation techniques might be sufficient for real-world deployment.

3.7 Discussions

3.7.1 Implementation issues

In a BRICK implementation, all sub-counter arrays (A_j) and index bitmaps (I_j) are fixed in size, and the number and size of buckets are also fixed. Consider the three level case shown in Table 11 with $k = 64$. Both lookup and increment operations can be performed with 10 memory accesses in total, 5 reads and 5 writes. For the bucket being read or updated, we first retrieve all bitmaps (I_j), bucket overflow status flag f_ℓ, and an index field t_ℓ to a full-size bucket in case a bucket overflow has previously occurred. All this information for a bucket can be retrieved in two memory reads with 64-bit words, the first word corresponds to I_1 with 64-bits, and the second word stores $I_2 = 3$ bits, the overflow status flag, and the t_ℓ (about 7 bits). If f_ℓ is not set, then we need up to three reads and writes to update the

three levels of sub-counter arrays. The updated index bitmaps and overflow status flags can be written back in two memory writes. If f_ℓ has been set, then we read directly from the corresponding entry in the full-size bucket indicated by t_ℓ for a lookup operation, avoiding the need to read the sub-counter arrays, hence requiring fewer memory acceses. Similarly, an increment operation for a counter that is already in a full-size bucket takes only one read and one write to update. If a bucket overflow occurs during an increment of a counter in a bucket, there is no need to access the last sub-counter array (otherwise, we wouldn't have an overflow). Therefore, we save two memory accesses at the expense of one write to the full-size bucket. With index bitmaps, overflow status flag, and full-size index field packed into two words, the worst case number of memory accesses is 10 in total, which permits updates in 20ns with a 2ns SRAM time, enabling over 15 million packets per second of updates.

BRICK is also amenable to pipelining in hardware. In general, the lookup or update of each successive level of sub-counter arrays can be pipelined such that at each packet arrival, a lookup or update can operate on A_1 while a previous operation operates on A_2, and so forth. This enables the processing of hundreds of millions of packets per second.

3.8 Conclusion

In this chapter, we have presented a novel exact active statistics counter architecture called BRICK (Bucketized Rank Indexed Counters) that can very efficiently store large arrays of variable width counters entirely in SRAM while supporting extremely fast increments and lookups. This high memory (SRAM) efficiency is achieved through a statistical multiplexing technique, which by grouping a fixed number of randomly selected counters into a bucket, allows us to tightly bound the amount of memory that needs to be allocated to each bucket. Statistical guarantees of BRICK are proven using a combination of stochastic ordering theory and probabilistic tail bound techniques. We also employed the rank-indexing data structure, similar to the rank-indexing technique in Chapter 2, to allow for fast random

access of every counter inside a bucket. Experiments with real-world Internet traffic traces show that our solution can indeed maintain large arrays of exact active statistics counters with moderate amounts of SRAM.

CHAPTER IV

DESIGN AND ANALYSIS OF ERROR ESTIMATING CODING: NEW SKETCHES, ESTIMATORS AND ANALYSIS FRAMEWORK

4.1 Problem Overview

Estimating the bit error rate (BER) in packets transmitted over wireless networks has been established as an important research problem in the seminal work of Chen *et al.* [13]. It was shown in [13] that, if the BER in packets can be accurately estimated, important operations in wireless networks such as packet re-scheduling, routing, and carrier selection can all be performed with greater efficiency. A simple yet effective technique, called error estimating codes (EEC), is proposed in [13] to help estimate this BER. Its basic idea is for the transmitter to send along with a packet a set of parity-check bits, each of which is the exclusive-or of a group of bits randomly sampled from the packet. These parity equations are designed in such a way that, by counting how many of them are violated after the packet transmission, the receiver can estimate, with low relative error, this BER.

Using an EEC of $O(\log n)$ bits for a packet n bits long, their technique guarantees that the estimated BER falls within $1 \pm \epsilon$ of the actual BER with probability at least $1 - \delta$, where ϵ and δ are tunable parameters that can be made arbitrarily small at the cost of increased constant factor in $O(\log n)$, the coding overhead.

A natural question to ask is whether EEC achieves the best tradeoff between space ($O(\log n)$) and estimation accuracy ((ϵ, δ) guarantee) in solving the BER estimation problem. In this dissertation work, we answer this question definitively from a very different alternative angle. While the EEC work looks at this problem from by and large a coding theoretic perspective, we can also look at it from a theoretical computer science perspective, modeling it as a so-called two-party computation problem as follows. Two parties

Alice and Bob each knows a (local) binary string x and y respectively, but Alice has no knowledge of y and vice versa. Alice and Bob are faced with the problem of computing the value of a function f acting upon the inputs x and y, often approximately. Intuitively, given any "sufficiently nontrivial" function f, for Alice and Bob to compute $f(x, y)$ together even approximately, either Alice has to tell Bob something about x or Bob needs to tell Alice something about y. The theory of two-party computation is concerned with how to evaluate $f(x, y)$ using as little communication (telling the other party about their local strings) between Alice and Bob as possible. Such a minimum amount of communication needed for the two-party computation of $f(x, y)$ is referred to as its *communication complexity*.

In the context of this work, two parties Alice and Bob are the transmitter and the receiver respectively. Alice knows the string (packet) x that is transmitted and Bob knows the string (packet) y that is received. The function f we would like to evaluate on (x, y) is clearly the Hamming distance between x and y, that is, $||x - y||_0$ (L_0 norm of the difference). We would like to find out the minimum amount of extra information (about x) that Alice needs to send to Bob, alongside with x, in order for Bob to approximately estimate $f(x, y)$. Techniques for (most) compactly encoding such extra information (about x) are referred to as *sketching algorithms* and the resulting encodings are called *sketches*.

Casting this BER estimation problem into the rich theoretical framework of two-party computation allows us to look much deeper into its underlying mathematical structures and obtain a set of new and better results. We can hence prove that the (randomized) communication complexity for the two-party computation of $||x-y||_0$ is $\Omega(\log n)$, where n is the length of the string x and y. In other words, Alice (the sender) needs to send to Bob (the receiver) a minimum of $\Omega(\log n)$ bits in order for Bob to approximately compute $||x - y||_0$. Since the number of overhead bits used in the EEC algorithm is indeed $O(\log n)$ [13], it matches this lower bound and is therefore asymptotically optimal.

A natural deeper and more important question following up is whether EEC has achieved the desired optimal space-accuracy tradeoff. In this dissertation work, we will answer this

question in Section 4.6 and demonstrate that EEC decoding is inefficient by deriving the amount of Fisher information contained in EEC codewords and showing that the variance of the estimator used in EEC encoding is much larger than the corresponding Cramer-Rao bound. In fact, we find that EEC codewords contain around one time more information than that is utilized by the estimator proposed in [13].

In order to overcome the inefficiency of the original EEC scheme, we then propose a new estimator that achieves a significantly higher estimation accuracy and is provably near-optimal by almost matching the Cramer-Rao bound. Our experiments will show that this new estimator allows us to reduce the coding overhead by as much as two to three times while achieving the same BER estimation accuracy. Another salient property of this new estimator is that its variance can be approximated by a closed-form formula, making it much easier to parameterize the EEC algorithm (i.e., to "tune") for optimal estimation accuracies (i.e., minimum variance) under various bit error models.

Successive to the improvement on the decoding part, we proceed to investigate whether there are inefficiencies with the encoding part of EEC. This question is, however, much harder to answer definitively because existing lower-bound techniques, rooted in the theory of communication complexity [41], only allow us to establish asymptotic space lower bounds such as the aforementioned $\Omega(\log n)$ bound.

Our first discovery on this actually still derives from two-party communication angle presented above. From that angle, our goal is actually is just measuring the the Hamming distance $||x-y||_0$ (the number of errors occurring during transmission), which is actually the same as $||x-y||_1$ (L_1 norm) and $||x-y||_2$ (L_2 norm)[1]. Various sketches have been proposed to compactly encode a (long) string x for the two-party computations of $||x-y||_0$, $||x-y||_1$, and $||x-y||_2$. Among them, we discover that the tug-of-war sketch [3] proposed for estimating the L_2 norms is most suitable for our purposes.

[1]Let x_i and y_i be the ith bit in x and y respectively, $i = 1, 2, ..., n$. Then the L_p norm of the difference vector $x - y$ is defined as $\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$.

However, the tug-of-war sketch *per se* is not yet the right solution to our problem for several reasons, which will be discussed in detail in Section 4.5. In short, we come on several nice techniques to make the solution competitive and we name it as "Enhanced Tug-of-War Sketch" (EToW).

Our next discovery, is that EEC and EToW can actually be viewed as different instances of a unified coding framework that we call generalized EEC (gEEC). In other words, gEEC can be parameterized into both EEC and EToW, and EEC can be viewed as a "degenerate" case of gEEC.

This generalization makes it easier for us to analyze and improve the designs of both EEC and EToW for two reasons. First, we need only design a single optimal decoder (i.e., estimator) for gEEC, which applies to both EEC and EToW, instead of one for each. This decoder is a Maximum Likelihood Estimator (MLE) with the Jeffreys prior, which in the case of EEC is the aforementioned estimator, and in the case of EToW performs better than a different estimator we developed for EToW in [30]. Second, the Fisher information formula derived for gEEC, which is in a closed form of matrix computations, applies to both EEC and EToW. Through this unified framework of gEEC, we found that some parameterization of gEEC (similar to EToW, but not needing the extra error detection bits) can contain around 25% more information than the pure EEC scheme. This information gain cannot be fully decoded through EToW's decoder, but is achievable by gEEC's decoder.

Note that, although the second discovery presented above looks comprehensive than EToW, EToW still has its special advantage in its simple decoding process.

4.2 Background, Preliminaries and Related Works

In this section, we will firstly overview the EEC problem and then provide the backgrounds related to the problem here.

4.2.1 Error Estimating Codes (EEC)

Error correcting coding [44] has been playing a fundamental and critical role in communication systems for more than fifty years, behind which is a philosophy that application and network can or should only deal with entirely correct data. However, in recent years, partially correct packets are found to be also useful in more and more designs, such as the designs with incremental redundancy ARQ[43], the designs that may collect and combine multiple partially correct packets[19, 36], the designs with forward error corrections[20], and audio/video communication where errors could be tolerated to some extent[57].

The seminal paper of Chen et al. [13] has brought to the fore the problem of (approximately) estimating the number of bit errors (correspondingly the bit error rate, BER) that has occurred to a packet during its transmission over a wireless network. In contrast of error correcting coding, Chen et al. [13] aim to use much smaller overhead while only providing light-weight function: merely estimating the number of bit errors without correcting them. A simple yet effective technique, called error estimating codes (EEC), is proposed in [13] to help estimate this BER. It has been shown in [13] that knowing this (approximate) bit error rate (BER) of a packet makes possible a host of advanced packet processing capabilities such as packet re-scheduling, routing, and carrier-selection schemes that can improve the (good) throughput of a wireless network in various ways. Compared to the previous solutions , which either uses indirect inference such as packet loss ratio and signal/noise ratio [29, 65], or needs special hardware support in the lower layer with the soft decoding capability [64], the EEC solution directly infers the BER from the packet and achieve better accuracy while also doesn't need special hardware support.

In EEC [13], the codeword for a packet consists of a set of $m = ab$ parity bits z_1, z_2, ..., z_m. They form a groups of size b each, $\{z_1, z_2, ..., z_b\}$, $\{z_{b+1}, z_{b+2}, ..., z_{2b}\}$, ..., $\{z_{(a-1)b+1}, z_{(a-1)b+2}, ..., z_{ab}\}$. Each parity bit z_i that belongs to group j (i.e., $(j-1)*b+1 \leq i \leq jb$) is calculated as the XOR of a set of $l_i = 2^j - 1$ bits uniformly (pseudo-)randomly sampled *with replacement* from the packet (viewed as a bit array). The size of groups (l_i's) are geometrically

distributed to maintain a good "estimation resolution" on a wide range of different BERs. In the following, we refer to each such group as a *level* to be consistent with the terms used in [13]. The codeword thus computed will be sent along with the packet to the receiver. Note the encoding scheme of EEC has some flavor of Low Density Parity Check (LDPC) codes although its parity check matrix is not strictly sparse as some of the rows can have as many as 2^a (including the parity check bit itself) ones in it, and in LDPC, no bit will be sampled more than once, which may happen in EEC due to its sampling with replacement nature.

Upon the receipt of a packet and its codeword (possibly with one or more bits flipped during transmission), the receiver will multiply them (viewed as a vector) by the same parity check matrix[2] and infer the BER from the outcome of this multiplication, which is often referred to as a *syndrome vector* in coding theory literature. From the syndrome vector, the inference algorithm (i.e., the decoder) used in [13] first decides on the group (i.e., level) of parity check bits that are expected to provide the best estimation accuracy. Then BER will be estimated *only* from the corresponding syndrome bits within that group.

The authors of [13] showed that that their scheme with 9 levels and less than 300 additional bits in total per packet would be able to well differentiate BER rate in range $[10^{-3}, 0.15]$, and it would work well in real-world wireless experiments and is a great enhancement. They also show that they provide a (ϵ, δ) bound analysis of the proposed scheme, i.e. they could guarantee at most ϵ relative error that failed with probability less than δ, where ϵ and δ are arbitrarily tunable parameters that determine the overhead cost of their algorithm. In total they need about $O(\log(n))$ overhead for an n bit packet to achieve error estimating rates within the threshold desired by target applications.

[2]Both the sender and the receiver know this matrix.

4.2.2 Randomized Approximation and Communication Complexity

Randomized approximation: The original EEC scheme, the tug-of-war sketch to be presented in Section 4.4.1, and the generalized EEC scheme to be presented in Section 4.7, are (ϵ, δ)-approximation schemes. An (ϵ, δ)-approximation algorithm is one that produces an estimate $\hat{X}$ for some quantity X with the guarantee that the absolute relative error $|\hat{X} - X|/X$ is at most ϵ with probability at least $1 - \delta$. It is assumed that $0 < \epsilon, \delta < 1$ are arbitrary constants that can be tuned by the designer of the algorithm. Typically, the cost of the scheme is dependent on these two parameters.

Communication complexity: Many of our lower bounds on this dissertation research make direct use of results from the communication complexity literature [41]. As mentioned before, communication complexity deals with the problem of determining the exact amount of communication needed between two parties to compute some function on their non-overlapping but jointly complete input. The communication complexity of a function at input size n is the largest number of bits that the two parties have to communication with each other using the *optimal* protocol for any input of size n. The basic communication complexity model can be generalized in many ways, two of which—randomization and one-round—appear in this thesis. The randomized communication complexity of a function is the communication complexity of an optimal randomized protocol that is correct with some positive constant probability (over random choices of the protocol). The one-round communication complexity of a function is the communication complexity of an optimal protocol in which Alice sends a single message to Bob, and Bob then computes the result of the function with no further communication. The problem of estimating BER clearly corresponds to the one-round model. Randomization is also allowed in our context and is actually used by both EEC and our scheme.

4.2.3 Fisher Information and Cramer-Rao Bound

In information theory and mathematical statistics, Fisher information quantifies the amount of information an observable parameterized random variable $X(\theta)$ carries about its unknown parameter θ. (Our description closely follows [16]; please consult it for more background on this topic.) In our context, the error estimating codewords and sketches correspond to random variable $X(\theta)$ and θ corresponds to the BER we would like to estimate in a received packet. The probability function of $X(\theta)$ takes the form $f(x;\theta)$. When θ is viewed as a constant and x as a variable, $f(x;\theta)$ is the probability density (or mass) of the random variable X conditional on the value of θ; When θ is viewed as a variable and x as a constant on the other hand, $f(x;\theta)$ is the likelihood function of θ, that is, the likelihood of the parameter taking value θ when the observed value of X is x. Fisher information of $X(\theta)$, which is a function of θ, is defined as

$$\mathbf{J}(\theta) \triangleq \mathbf{E}_\theta\left[\frac{\partial}{\partial\theta}\log f(X;\theta))\right]^2. \tag{15}$$

Fisher information $\mathbf{J}(\theta)$ is an important quantity because it determines the minimum variance achievable by any unbiased estimator of θ given an observation of $X(\theta)$, through the Cramer-Rao lower bound (CRLB):

$$\mathbf{MSE}[\hat{\theta}] = \mathbf{Var}[\hat{\theta}] \geq \frac{1}{\mathbf{J}(\theta)}. \tag{16}$$

If a biased estimator of θ with bias $b(\theta)$ is used instead, we have a slightly different inequality:

$$\mathbf{MSE}[\hat{\theta}] \geq \frac{(1 + b'(\theta))^2}{\mathbf{J}(\theta)} + b(\theta)^2. \tag{17}$$

While it is possible for a biased estimator to "beat" the Cramer-Rao bound for unbiased estimators (Formula (2)) when θ takes certain values (over which $b'(\theta)$ takes negative values), that biased estimator is not a clear winner since bias comes at a cost and may not be desirable to many applications.

In the context of this work, where the goal is to measure the "scale" of the bit error rate θ, the statistics of the relative error $\frac{\hat{\theta}-\theta}{\theta}$ and the log-difference $\log\hat{\theta} - \log\theta$ are more important. In particular, we prefer to use the statistics of the log-difference rather than the relative ratio to evaluate the performance of an estimator, since it assigns higher penalty to large deviations, making the comparison fairer for this application setting. For example, suppose the real value of θ is 0.1, and three large-deviation estimates are 0.05, 0.19 and 0.01, then the penalty for the last one will be much larger if measured by $\log\hat{\theta}$'s statistics. However, when measured by the relative error, the penalties of the latter two are the same and just around twice of the first.

The C-R bound for both $\frac{\hat{\theta}-\theta}{\theta}$ and $\log\hat{\theta} - \log\theta$ can be derived from (16) and the results are as follows:

$$\mathbf{Var}\left[\frac{\hat{\theta}-\theta}{\theta}\right] = \mathbf{Var}\left[\frac{\hat{\theta}}{\theta}\right] \geq \frac{1}{\theta^2 J(\theta)}. \tag{18}$$

$$\mathbf{Var}\left[\log\hat{\theta} - \log\theta\right] = \mathbf{Var}\left[\log\hat{\theta}\right] \geq \frac{1}{\theta^2 J(\theta)}. \tag{19}$$

Interestingly, they are bounded by the same value, $\theta^2 J(\theta)$. The second inequality (19) is derived by a transformation from (16). In other words, $\theta^2 J(\theta)$ is the Fisher information of $\log\theta$. We will use $\theta^2 J(\theta)$ frequently throughout the chapter since it will directly determine the bound of the relative error/log difference.[3]

The maximum likelihood estimator (MLE) of θ, defined as $\hat{\theta}_{MLE} \triangleq \arg\max_\theta\{f(x;\theta)\}$, is known to be asymptotically normal (denoted as $N(*,*)$) under certain regularity conditions [42] and has the following distribution:

$$\hat{\theta}_{MLE} \sim N\left(\theta, \frac{1}{tJ(\theta)}\right), \tag{20}$$

where t here denotes the number of repeated independent experiments and $J(\theta)$ denotes the Fisher information contributed from each experiment. Hence the Cramer-Rao lower bound is (asymptotically) reached by the MLE.

[3] There is another concept called relative Fisher information established in information theory which is not related to anything here.

Fisher information and Cramer-Rao bound analysis has been used in a few previous works on network measurement in the literature. It has been used by Ribeiro *et al.* [54] to derive the minimum number of samples needed for accurately estimating flow size distributions from outputs of a packet sampling process (e.g., sampled Cisco NetFlow). They also proposed an unbiased MLE estimator for this estimation problem that empirically matches the Cramer-Rao bound. This work was followed up in [60] by Tune *et al.* who demonstrated through Fisher information analysis that samples collected by flow sampling, which is much more expensive computationally, are more information-rich, in terms of Fisher information per bit, than packet sampling. They then proposed a new hybrid sampling technique called *dual sampling* that combines the advantages of both flow and packet sampling. Fisher information analysis is also used in recent work [61] to compare the information-richness of the samples collected by a few packet sampling and sketching techniques for the purpose of estimating flow size distributions.

4.2.4 Data Streaming and Communication Complexity

One of the major technical contribution that we make in this thesis is to adapt sketching algorithms from the field of data streaming to this problem. Data streaming is a well-studied area with a rich literature [52]. In the data streaming model, the input is provided as a long stream of updates in which only a single pass is allowed over the stream and the memory and time of the algorithm is heavily constrained (in particular much smaller than the size of the input). The connection to this problem is that there are many streaming algorithms that can be used to compute the *difference* (or distance) between two streams, and the summaries of these algorithms (called sketches) are what we can use as overhead bits for this problem. We tried several different sketching algorithms, including the count-min sketch [15], the Flajolet-Martin (FM) sketch [25], the stable distribution sketch [34], before settling on our variation on the tug-of-war sketch [3]. The tug-of-war sketch was originally suggested by Alon et al. [3] for estimating the second frequency moment of a data

stream. The governing characteristic of this sketch, that we make use of in this thesis, is that it is a random projection of the input, thereby allowing for deletions from the received packet. This sketch was modified for measuring the $L1$ distance of streams by Feigenbaum et al. [23].

As for the proof of the lower bounds, our lower bounds make direct use of known results for the communication complexity [41] of the Hamming distance problem. Our main lower bound, showing that $\Omega(\log(n)/\epsilon^2)$ overhead is necessary is a consequence of the communication complexity result from [38].

4.2.5 Definitions and notes of Notations

For convenience, all major notations that will be used are summarized in the following table for future reference

4.3 Lower Bounds

We next show lower bounds for the BER estimation problem, demonstrating the optimality of our algorithms. In Sec. III.A, we show that deterministically estimating the error rate (i.e., without the use of randomization) requires the coding scheme to use $\Omega(n)$ bits of overhead even when we allow the estimate to err by over 10% from the actual value. Similarly, we show in Sec. III.B that randomization alone cannot produce the exact BER estimation using a well-known result from the area of communication complexity. Based on these two results, one can see why we can only approximate the result with high probability. Finally, we show in Sec. III.C why the $O(\log n)$-bit sketch our algorithms use is necessary.

4.3.1 Why Randomization Is Needed

Theorem 1. *Any error-estimating scheme that estimates the number of the bits in an n-bit packet that change during transmission to within $n/8$ must use $\Omega(n)$ overhead bits.*

Proof. Let n be divisible by 8 (the argument works for all n with some slight modifications). It is known that there exists a family G of $2^{\Omega(n)}$ subsets of $\{1, 2, 3, \ldots, n\}$ such that (i) each set

Notation	Definition	Schemes	Note
		Parameters	
n	binary length of the original packet (data size)	All schemes	In practice, typically less than 12,000 (1500bytes)
m	binary length of the sketch bits for EEC purpose	All schemes	Typically 80 to 300 bits
l_i	sampling group size of the ith EEC-bit/sub-sketch	EEC/gEEC	Dependant on application.
l	sampling group size of each sub-sketch	EToW	Dependant on application.
k	the binary length of the ith sub-sketch sent	EToW	Typically 3-5
k_i	the binary length of the ith sub-sketch sent	gEEC	Typically 1-6
r	the binary length of the checking bits of each sub-sketch sent	EToW	Typically 1-2
		Variables	
θ	bit error rate (BER)	All schemes	We are estimating the BER of each packet, not BER of the channel.
$\hat{\theta}$	Estimator of θ	All schemes	
$\overrightarrow{b}$	Binary vector of the data bits	ToW, EToW and gEEC	n-entries(bits), typically defined in $\{-1,1\}^n$, unless specifically designated
$\overrightarrow{s_i}$	Binary random vector for the ith sub-sketch	EToW,gEEC	n-entries(bits), defined in $\{-1,1\}^n$
z_i or $\overrightarrow{z_i}$	the ith EEC-bit/sub-sketch sent	EEC/EToW,gEEC	
$\overrightarrow{b'}$	the data bits received	All schemes	1-bit in EEC, k-bit in EToW, k_i-bit in gEEC
z'_i or $\overrightarrow{z'_i}$	the ith EEC-bit/sub-sketch calculated from $\overrightarrow{b'}$	EEC/EToW,gEEC	The same size as above
$\check{z}_i$ or $\overrightarrow{\check{z}_i}$	the ith EEC-bit/sub-sketch received	EEC/EToW,gEEC	The same size as above
q_i	the checking bits for i^{th} sub-sketch sent	EToW	r-bits
$\check{q}_i$	the checking bits for i^{th} sub-sketch received	EToW	The same size as above

Table 13: Definition of symbols for Error Estimating Problem

in G has cardinality exactly $n/4$, and (ii) every pair of sets in G have at most $n/8$ elements in common. The existence of such a family can be shown using the probabilistic method, but this is omitted here for brevity.

Let us assume for a contradiction that there exists a deterministic sketch of size less than $\Omega(n)$ bits that allows the computation of the Hamming distance between the original and transmitted codewords within an error of less than $n/8$. Consider what happens when we sketch all the codewords formed by the characteristic vectors of the sets in G. Since the sketch size is less than $\log(|G|) = \Omega(n)$, by the pigeonhole principle we know that two of the sets, say g_1 and g_2, in G must result in the same sketch value, making them indistinguishable. The Hamming distance between these two sets is at least $n/8 + n/8 = n/4$. As a result, since the sketch cannot distinguish between the cases when the original codeword and the transmitted codeword correspond to g_1 and g_1, versus when they correspond to g_1 and g_2, respectively, one of these two cases must have an error of at least $n/8$. □

4.3.2 Why Approximation is Needed

Theorem 2. *Any error-estimating scheme that computes the exact number of bits in an n-bit packet that change during transmission with probability at least $3/4$ must use $\Omega(n)$ overhead bits.*

Proof. For this result, we use the communication complexity of the Set Disjointness problem. It is known that for two parties to compute whether their subsets of $\{1, 2, 3, \ldots, n\}$ have any elements in common requires $\Omega(n)$ communication, even when randomization (with $1/4$ failure probability) is allowed [37].

Assume for a contradiction that there is a randomized sketch using less than $\Omega(n)$ bits that can be used to compute the Hamming distance between the original and transmitted codewords exactly. We use this to create the following protocol for Set Disjointness. Alice uses the sketch to summarize the characteristic vector of her set and sends the sketch (less than $\Omega(n)$ bits) and the number of elements in her set ($\log n$ bits), call it n_a, to Bob. Bob

can now use this information to compute the Hamming distance (call it h), the number of elements in his set (call it n_b), and then compute the size of the intersection of his and Alice's set as $(n_a + n_b - h)/2$. This is a one-round randomized protocol to compute the size of the intersection of Alice and Bob's sets, and hence must use $\Omega(n)$ communication, contradicting our assumption about the size of the sketch. □

4.3.3 Randomized Approximation

We now use a lower bound in [38] to show that the asymptotic complexity of the tug-of-war sketch and the original EEC scheme are optimal in terms of their dependence on n and ϵ, the relative error bound. The lower bound result we use is as follows:

Theorem 3 ([38, 66])**.** *The randomized one-round two-party communication complexity of approximating the Hamming distance of the n-bit vectors of two parties up to a relative error of ϵ with constant probability is at least* $\Omega(\log(n)/\epsilon^2)$.

The reduction is the same as the last, and a lower bound of $\Omega(\log(n)/\epsilon^2)$ on the sketch size follows.

4.4 Tug-of-War Sketch for Error Estimating Coding

4.4.1 The sketch

In this section, we briefly describe and analyze the plain vanilla tug-of-war sketch [3] in the context of error estimating coding, under the assumption that the sketch per se is not subject to bit errors during transmission. The tug-of-war sketch of a bit array (packet) b is comprised of a constant number c of counters (c is determined by the desired error guarantees) that are maintained using the same update algorithm (with possibly different update values) and is sent to the receiver alongside with b. After the execution of these

update algorithms, each counter contains the inner product of the bit array[4] $\overrightarrow{b}$ with a pre-defined pseudorandom vector $\overrightarrow{s} \in \{+1, -1\}^n$. Note the actual update algorithm is not shown here because it is not relevant to our context; Only its "net effect" after execution is.

As shown in the following algorithm, upon the receipt of the transmitted bit array $\overrightarrow{b'}$ and the sketch (assumed to have no bit error during transmission), the receiver computes the c inner products using the received packet (possibly with bit errors) $\overrightarrow{b'}$, takes the difference between them and the counters in the sketch sent along the packet, and squares the result. Each of these results is now an unbiased estimate (proved in [3]) of the Hamming distance between the original and the transmitted packets, and can be averaged to give an accurate estimate of the Hamming distance[5], d. The details are given in Algorithm 2.

Algorithm 2: The tug-of-war sketch for EEC.

1 SKETCH–CREATION($\vec{b}$)
  Input $\vec{b}$: original data bits vector.
  Output z: the sketch encoding $\vec{b}$.
  pre-compute random vectors $\vec{s}_{j,1\le j\le c} : [n] \to \{-1, 1\}$
  for $j = 1$ to c **do**
    $z_j := (\vec{b} \cdot \vec{s}_j)/2$
  end for
  return $z = \langle z_1, \ldots, z_c \rangle$

DISTANCE–ESTIMATION($\vec{b}', z$)
  Input $\vec{b}'$: received data bits vector, z: received sketch.
  Output $\hat{p}$: the estimated error rate.
  pre-compute random vectors $\vec{s}_{j,1\le j\le c} : [n] \to \{-1, 1\}$
  for $j = 1$ to c **do**
    $X_j := (z_j - \vec{b}' \cdot \vec{s}_j/2)^2$
  end for
  return $\hat{\theta}=\frac{1}{n}$average$(X_1, \ldots, X_c)$

[4]Here $\overrightarrow{b}$ is the vector representation of the packet b, where '0's have been converted to '-1's as discussed earlier.

[5]Note that this is a simplified form of the tug-of-war sketch proposed in [3]. The original version reduced the dependence on δ to $\log(1/\delta)$ by computing the average of $O(\frac{1}{\epsilon^2})$ estimators and then finding the median of $O(\log(1/\delta))$ such groups, at the cost of a larger constant multiplicative factor. For simplicity of the analysis in the following section, we omit this asymptotic improvement here.

4.4.2 Analysis

We now show that this estimator (the average of component random variables X_1, X_2, ..., X_c) has low variance. To compute the variance of each component X_j, we first compute

$$
\begin{aligned}
\mathbf{E}[X_j^2] &= \mathbf{E}\left[(\sum_{b_i \neq b_i'} b_i s_j[i])^4 \right] \\
&= \mathbf{E}\left[\sum_{b_i \neq b_i'} (b_i s_j[i])^4 \right] + \mathbf{E}\left[6 \sum_{b_i \neq b_i' \wedge b_k \neq b_k' \wedge i \neq k} (b_i s_j[i])^2 (b_k s_j[k])^2 \right] \\
&= d + 6d(d-1)/2 \\
&= 3d^2 - 2d, \qquad (21)
\end{aligned}
$$

Substituting this into the expression for the variance of X_j, we obtain

$$
\begin{aligned}
\mathbf{Var}[X_j] &= \mathbf{E}[X_j^2] - (\mathbf{E}[X_j])^2 \\
&= 3d^2 - 2d - d^2 \leq 2d^2.
\end{aligned}
$$

Although the variance of a single component X_j may look large (giving a standard deviation larger than d itself), averaging c of them reduces it by a factor of c. Using Chebyschev's inequality then allows us to bound the failure probability arbitrarily small as well (depending solely on how large we allow c to get). More concretely, if we pick $c = \frac{2}{\epsilon^2 \delta}$, then by Chebyschev's inequality we get that the estimate $\hat{d}$ from the above algorithm has the guarantee

$$
\Pr[|\hat{d} - d| \geq \epsilon d] \leq \frac{\mathbf{Var}[\hat{d}]}{\epsilon^2 d^2} \leq \frac{2d^2}{c\epsilon^2 d^2} = \delta.
$$

Correspondingly, the relative error of the final estimate of $\hat{\theta} = \frac{1}{n}\hat{d}$ would also satisfy the $\epsilon - \delta$ bound:

$$
\Pr[|\hat{\theta} - \theta| \geq \epsilon\theta] = \Pr[|\hat{d} - d| \geq \epsilon d] \leq \delta.
$$

The total overhead of this scheme is that of sending $c = \frac{2}{\epsilon^2 \delta}$ (a constant, independent of n) counters, each of which contains a number in the range $[-n, n]$. Hence, the asymptotic cost is $O(\log n)$ bits.

4.4.3 Cost and Overhead for EEC applications

In this section, we perform a rough estimate of the total size of the sketch if the plain vanilla tug-of-war sketch is used directly for EEC applications. This is needed for us to compare it with our enhanced sketch, to be described in the next section. To allow for a fair comparison, here we no longer assume the sketch is immune from bit errors during transmissions. The total size of the sketch is determined by three factors: the number of counters c, the size of each counter (denoted as k), and the number of extra bits needed to protect the sketch. Since our enhanced sketch uses the same number of counters, we only need to discuss the second and third factors here for comparison purposes.

To estimate the size of each counter, let us assume that the (maximum) length of the packet is 1500 bytes = 1.2×10^4 bits. In the worst case $\log_2(n) = \log_2(12000) \approx 14$ is needed per counter, since $\max(\vec{b} \cdot \vec{s}/2) = n/2$ and $\min(\vec{b} \cdot \vec{s}/2) = -n/2$. However, the value of each counter in the sketch, which is a random variable, has its probability densities concentrated around its mean 0, since $\vec{b} \cdot \vec{s}$ is the sum of n i.i.d. Bernoulli random variables $b_i s_i$, each of which takes value $+1$ or -1 with equal probability 0.5. We calculate the tail probability of the resulting Binomial distribution and find $\Pr(|Z| > 255) \approx 3 \times 10^{-6}$. Therefore, if we truncate each counter to 9 bits (including one sign bit, since z could be positive or negative) from 14, we risk overflowing it with probability 3×10^{-6}.

A lower bound of the number of additional bits needed to protect the sketch can be estimated using information theory as follows. Suppose the bit errors are symmetric (equal probability in flipping 1 to 0 and the other way around) and random, the amount of information brought by each bit received is:

$$I(\theta) = 1 + \theta \lg \theta + (1 - \theta) \lg(1 - \theta).$$

Therefore, the final size of the sketch, including all the protection bits, needs to be at least $I(\theta)^{-1}$ times larger than the original sketch. For example, when the error rate is 0.15, the blowup factor $I(0.15)^{-1}$ is equal to 2.56.

Summing all above factors together, our back-of-envelope conclusion is: when the overheads of extra protection is not counted in, roughly $16 \times 9 = 144$ would be able to deliver comparable performance. Compared with the $9 \times 32 = 288$ bits needed by the EEC scheme, using tug-of-war sketch would gain considerable reduction on the overhead while the estimation procedure would be very simple. However, when the extra protection cost is counted in, the tug-of-war sketch would be no longer of too much advantage.

In the next section, we would present the enhanced tug-of-war sketch, which could reduce the size of each counter by roughly another half and protect the sketch with small overhead, while still delivering comparable performance.

4.5 Enhanced Tug-of-war Sketch (EToW): Scheme and Analysis

As mentioned before, we enhance the vanilla tug-of-war sketch in the following three ways to achieve better space-accuracy tradeoffs and to be able to handle bit errors that may occur to the sketch. As shown in Algorithm 3, our enhanced sketch contains a number of important improvements.

First, in some BER estimation scenarios, we need only know whether the BER falls into a certain interval like $[2^{-i}, 2^{-i+1}]$, as suggested in [13], rather than the exact BER value. In some others, only a rough BER estimation is called for. The vanilla tug-of-war sketch can be an overkill for both types of scenarios at the cost of an unnecessarily large sketch size. Adding to the problem is the fact that a larger sketch is more susceptible to bit errors during transmission and requires stronger protections which we can ill afford. Our solution is to combine sampling with sketching, in which we randomly sample l bits of the packet and sketch only these l bits according to Algorithm 3. A smaller l value leads to a smaller counter size and hence a smaller sketch size, at the cost of lower BER estimation accuracy due to higher sampling error. By adjusting this parameter l, we can minimize the sketch size needed to achieve a desired level of accuracy. The analysis needed for tuning this parameter for best size-accuracy tradeoffs is presented later in Sec. 4.5.1.

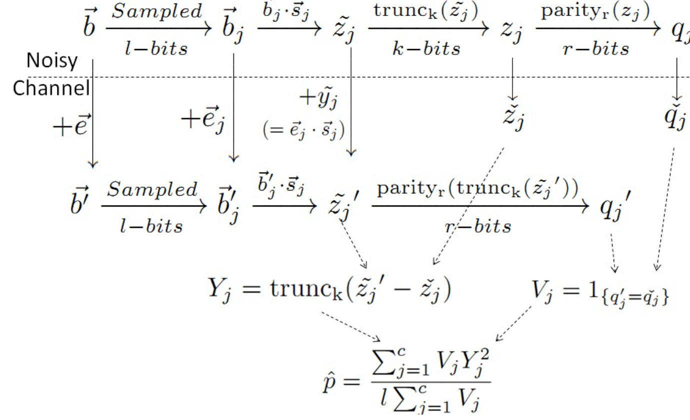


Figure 16: Overview of Enhanced Tug-of-War (EToW) Sketch (Algorithm 3)

Second, as we mentioned before, since the counter value (a random variable) stays close to its mean 0 with high probability, we may use fewer (say k) bits to store it without causing an "overflow" most of time. We refer to this enhancement as "statistical truncation", or *truncation* in short, for the lack of a better word. Even when an overflow (at either the sender or the receiver side) does happen (albeit with a small probability), its impact on estimation is small because with high probability truncation happens at both sides, in which case their difference remain the same as when there are no truncations. The impact of statistical truncation on the estimation accuracy will be analyzed in Sec. 4.5.2.

Finally, as previously mentioned, the sketch is not immune to bit errors during transmission and requires some protection. In our scheme, each counter (5 bits long) will be protected by a parity bit (an overhead ratio of 20%). Any counter that fails the parity check will be considered corrupted and will not be included in the estimation. The rationale for this choice will be explained in Sec. 4.5.3.

The parameters of the enhanced tug-of-war sketch in the following analyses are as follows. We let c be the total number of counters, l the number of bit positions sampled, k the length of each counter, and r the length of the parity bits of each counter. For convenience, we denote the sketch by Sketch(c, l, k, r). The total transmission cost is $c(k + r)$.

Algorithm 3: Enhanced tug-of-war sketch with parameters (c, l, k, r).

1 SKETCH–CREATION($\vec{b}$)

    Input $\vec{b}$: original data bits vector.
    Output z: the sketch of $\vec{b}$.
    pre-compute l-bit-long vectors $\vec{s}_{j,1\le j\le c} : [l] \to \{-1, 1\}$
    for $j = 1$ to c **do**
        l-bits-long vector $\vec{b}_j$: sampled with replacement from $\vec{b}$
        Random projection: $\tilde{z}_j := (\vec{b}_j \cdot \vec{s}_j)/2$
        k-bits-long truncated projection: $z_j := \mathrm{trunc_k}(\tilde{z}_j)$
        r-bits-long parities $q_j := \mathrm{parity_r}(z_j)$
    end for
    return $c(k + r)$-bits long sketch $z = \langle z_1, \ldots, z_c\rangle\langle q_1, \ldots, q_c\rangle$

DISTANCE–ESTIMATION($\vec{b}', \check{z}$)

    Input $\vec{b}'$: received data bits vector, $\check{z}'$: received sketch.
    Output $\hat{\theta}$: estimate of the error rate θ.
    pre-compute l-bit-long vectors $\vec{s}_{j,1\le j\le c} : [l] \to \{-1, 1\}$
    for $j = 1$ to c **do**
        l-bits-long vector $\vec{b}'_j$: sampled with replacement from $\vec{b}'$
        (with the same hash seed pre-configured.)
        Random projection: $\tilde{z}_j{}' := (\vec{b}'_j \cdot \vec{s}_j)/2$
        Estimation $Y_j := \mathrm{trunc_k}(\tilde{z}_j{}' - \check{z}_j)$,$X_j = Y_j^2$
        Check parities $V_j := 1_{\{q_j = \mathrm{parity_r}(\mathrm{trunc_k}(\tilde{z}_j{}'))\}}$
    end for
    return $\hat{\theta} = \frac{\sum_{j=1}^{c} V_j X_j}{l \sum_{j=1}^{c} V_j}$ as the estimation of error rate θ.

4.5.1 Analysis of the effect of sampling

Here we provide a simple analysis of the "sampled tug-of-war sketch" and show the benefits of sampling. In this analysis, we assume that no truncation is used and that no bit error happens during the transmission of the sketch. The value of a full (un-truncated) counter (a random variable) is denoted as $\tilde{z}$. Note that we use $\tilde{X}_j := \frac{1}{l}(\tilde{z}_j - z_j)^2$ to estimate the error rate $\theta = d/n$.

Note that the Hamming distance between the two sampled segments, denoted by D, is no longer a constant. From Section 4.4, we have $\mathbf{E}[X_j^2|D_j] = D_j$ and $\mathbf{Var}[X_j^2|D_j] = 3D_j^2 - 2D_j$. Since the l bits are sampled with replacement, D follows a binomial distribution, $\mathbf{E}[D] = lp$ and $\mathbf{Var}[D] = lp(1-\theta)$, where θ is the error rate.

The new estimator of $\hat{\theta} = \frac{1}{l}\tilde{X}_j$ is still unbiased:

$$\mathbf{E}[\hat{\theta}] = \frac{1}{l}\mathbf{E}[\tilde{X}_j] = \frac{1}{l}\mathbf{E}[\mathbf{E}[\tilde{X}_j|D]] = \frac{1}{l}\mathbf{E}[H] = \theta.$$

As for variance, we have

$$\begin{aligned}
\mathbf{Var}[\hat{\theta}] &= \frac{1}{l^2}\mathbf{Var}[\tilde{X}_j] \\
&= \frac{1}{l^2}(\mathbf{Var}[\mathbf{E}[\tilde{X}_j|D]] + \mathbf{E}[\mathbf{Var}[\tilde{X}_j|D]]) \\
&= \frac{1}{l^2}(\mathbf{Var}[D] + \mathbf{E}[2D^2 - 2D]) \\
&= \frac{1}{l^2}(lp(1-\theta) + 2lp(1-\theta) + 2l^2\theta^2 - 2l\theta) \\
&= \theta^2(2 + \frac{1}{l\theta} - \frac{3}{l}) < \theta^2(2 + \frac{1}{l\theta}).
\end{aligned} \tag{22}$$

Compared with (21), the relative variance of the sampled tug-of-war sketch is bounded by 2 plus an additional term $\frac{1}{pl}$. This means only when θ is $\Omega(\frac{1}{l})$ will the estimator achieve good performance. We can clearly observe this difference in Fig. 17. The sampled sketch with $l = 512$ performs very close to the original tug-of-war sketch when $\theta > 10^{-2}$, and much worse when $\theta < 10^{-3}$. From a practical perspective, this is exactly what we have intended. In typical application scenarios exemplified by [13], very accurate estimation (constant relative error) for very small bit error rates is not needed. In other words, the

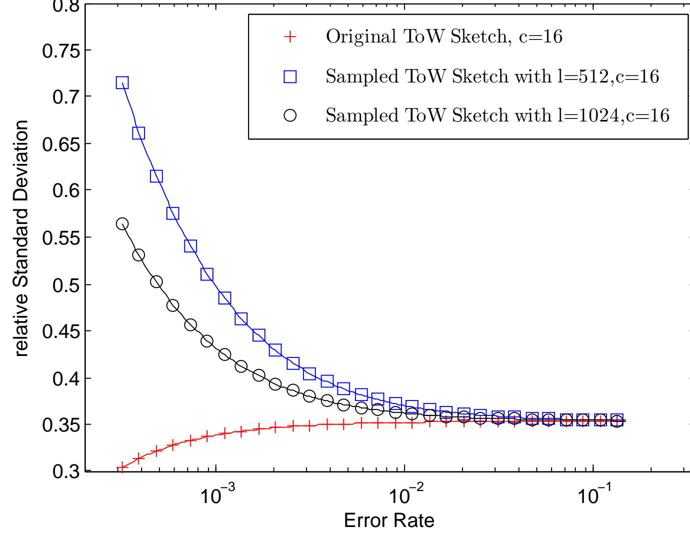


Figure 17: Comparison of the variance of the sampled tug-of-war sketches and the original one, with $c = 16$.

sampled tug-of-war sketch enables users to make the best use of bits for the applications we are interested in.

As discussed in Sec. 4.3, the communication complexity bound for measuring the Hamming distance is $\Omega(\log n)$. Both the original tug-of-war sketch and the EEC sketch match this bound. The sampled tug-of-war sketch uses only $O(\log l)$ bits, where l is a constant parameter that can be much less than n. This does not contradict our lower bound, however, since it does not deliver the target estimation accuracy ((ϵ, δ)-approximation) for inputs with certain BER parameter settings.

4.5.2 Analysis of truncation and sampling together

In this section, we analyze the impact of truncation on the overall estimation accuracy. The operation of truncating a counter value to a k-bit number (including one bit needed to represent the sign) can be formalized as follows:

$$z = \mathrm{trunc_k}(\tilde{z}) \equiv \tilde{z} (\mathrm{mod}\ 2^k), z \in [-2^{k-1}, 2^{k-1} - 1]. \tag{23}$$

As shown in Algorithm 3 and Figure 16, we defined $\tilde{Z}_j' = \vec{b}_j' \cdot \vec{s}_j$ and $Y_j = \mathrm{trunc_k}(\tilde{Z}_j' - Z_j)$.

We can find the following relationship between Y_j and the original (un-truncated) sketch values $\tilde{Z}_j$ and $\tilde{Z}_j{}'$ as follows:

$$
\begin{aligned}
Y_j = &\ \text{trunc}_\text{k}(\tilde{Z}_j{}' - Z_j) \quad , Y \in [-2^{k-1}, 2^{k-1} - 1] \\
\equiv &\ \tilde{Z}_j{}' - Z_j \quad (\text{mod } 2^k) \\
\equiv &\ \tilde{Z}_j{}' - \tilde{Z}_j \quad (\text{mod } 2^k), \text{ due to (23)} \\
\equiv &\ \text{trunc}_\text{k}(\tilde{Z}_j{}' - \tilde{Z}_j) \quad (\text{mod } 2^k)
\end{aligned}
\tag{24}
$$

In the following, we derive the distribution, expectation and variance of each estimator $\hat{\theta}_j = \frac{1}{l}X_j = \frac{1}{l}Y_j^2 = \frac{1}{l}\text{trunc}_\text{k}(\tilde{Z}_j{}' - \tilde{Z}_j)^2$.

Note Y_i, $i = 1, 2, ..., c$, are i.i.d. discrete random variables. Let Y be an arbitrary Y_i. In the following we will derive the probability mass function (PMF) of Y so that we can analyze the impact of truncation on our estimation accuracy. Since Y takes values on exactly 2^k integer values $\{-2^{k-1}, -2^{k-1}+1, ..., 2^{k-1}-1\}$, its PMF can be determined by a 2^k-dimensional vector $\vec{\gamma}(\theta, l) \equiv \langle \gamma_{-2^{k-1}}(\theta, l), \gamma_{-2^{k-1}+1}(\theta, l), ..., \gamma_{2^{k-1}-1}(\theta, l)\rangle$ where $\gamma_i(\theta, l) \equiv \Pr(Y = i|\theta, l)$, $i \in [-2^{k-1}, -2^{k-1} + 1, ..., 2^{k-1} - 1]$. Note that each scalar γ_i is a function of the error rate θ and the number of bits sampled l. We show that $\vec{\gamma}(\theta, l)$ can be computed from the following recurrence relation.

Lemma 4.

$$
\vec{\gamma}(\theta, l)_{1\times 2^k} = \vec{\gamma}(\theta, l-1)_{1\times 2^k}\mathbf{M}(\theta)_{2^k\times 2^k},
\tag{25}
$$

where

$$
\mathbf{M}(\theta) = \begin{bmatrix}
1-\theta & \theta/2 & & \cdots 0 \cdots & & & \theta/2 \\
\theta/2 & 1-\theta & \theta/2 & & \cdots 0 \cdots & & \\
& \theta/2 & 1-\theta & \theta/2 & & \cdots 0 \cdots & \\
& & \cdots & & & \cdots & \\
& \cdots 0 \cdots & & & \theta/2 & 1-\theta & \theta/2 \\
\theta/2 & & \cdots 0 \cdots & & & \theta/2 & 1-\theta
\end{bmatrix}_{2^k\times 2^k}
\tag{26}
$$

Proof. Let $\vec{e} = \langle e_1, e_2, \cdots, e_l \rangle$ and $\vec{s} = \langle s_1, s_2, \cdots, s_l \rangle$. We define the following interim random variables Y_j, $j = 0, \cdots, l$:

$$Y_j = \sum_{k=1}^{j} e_k s_k. \tag{27}$$

Clearly, $X = Y_l (\text{mod } 2^k)$. Due to the sampling with replacement policy, the increment in each step $\Delta Y_j = Y_{j+1} - Y_j = e_{j+1} s_{j+1}$ is independent of every other. Hence the random variables $\{Y_j\}_{0 \le j \le l}$ make up a Markov chain. In each step, with probability $1 - \theta$ an unchanged bit is selected and hence $\Delta Y_j = 0$; with probability θ a changed bit is selected, and half of these increments are $+1$ and the other half -1 since s_k is uniformly at random from $\{-1, 1\}$.

Hence the distribution of ΔY_j is:

$$\Delta Y_j = \begin{cases} -1 & \text{with prob } \theta/2, \\ 0 & \text{with prob } 1 - \theta, \\ 1 & \text{with prob } \theta/2. \end{cases} \tag{28}$$

Mapping $\{Y_j\}_{0 \le j \le l}$ into the finite field Z_{2^k}, formula (56) becomes the transition matrix, which is the circular matrix M defined in (53). □

To allow for efficient matrix computations, $\mathbf{M}(\theta)$ can be diagonalized as follows.

$$\mathbf{M}(\theta) = \frac{1}{K} \mathbf{\Omega}' Diag(d_0, d_1, \ldots, d_{K-1}) \mathbf{\Omega},$$

where $\Omega = \{\omega_{ik}\}$, $\omega_{ik} = \exp\left(\frac{2\pi ikj}{K}\right)$, j is the imaginary unit, and $d_i = 1 - 2\sin^2(2i\pi/K)\theta$.

Considering that $\vec{\gamma}(\theta, 0) = [0, \cdots, 0, 1, 0, \cdots]_{1 \times K}$, where 1 appears at the $(2^{k-1} + 1)^{th}$ position, we have

$$\begin{aligned} \vec{\gamma}(\theta, l) &= [0, \cdots, 1, 0, \cdots] \mathbf{M}(\theta)^l \\ &= \frac{1}{2^k} [0, \cdots, 1, 0, \cdots] \mathbf{\Omega}' Diag(d_0^l, d_1^l, \ldots, d_{K-1}^l) \mathbf{\Omega} \\ &= \frac{1}{2^k} [d_0^l, -d_1^l, \ldots, d_{2^k-2}^l, -d_{K-1}^l] \mathbf{\Omega}. \end{aligned} \tag{29}$$

Finally, we can calculate the expectation and the variance of Y^2 from $\vec{\gamma}(\theta, l)$ as follows:

$$\mathbf{E}[Y^2] = \vec{\gamma}(\theta, l)\boldsymbol{\beta}_2 \tag{30}$$

$$\mathbf{Var}[Y^2] = \vec{\gamma}(\theta, l)\boldsymbol{\beta}_4 - (\vec{\gamma}(\theta, l)\boldsymbol{\beta}_2)^2, \tag{31}$$

where $\boldsymbol{\beta}_i = [(-2^{k-1})^i, (-2^{k-1}+1)^i, ..., (2^{k-1}-1)^i]$.

After summing up all c counters in the sketch, we finally arrive at the Mean Squared Error of the estimator,

$$\begin{aligned} l^2\mathbf{MSE}[\hat{\theta}] &= \mathbf{E}[(\frac{1}{c}\sum Y_i^2 - pl)^2] \\ &= \frac{1}{c}\mathbf{Var}[Y^2] + (\mathbf{E}[Y^2] - pl)^2. \end{aligned} \tag{32}$$

Since $\frac{1}{l}\tilde{Y}_j^2$ is unbiased and $|\mathrm{trunc_k}(x)| \leq |x|$, $\frac{1}{l}Y_j^2$ will be (slightly) negatively biased. Note the aforementioned Chebyshev's inequality still holds for the mean squared error, while it does not for the variance when the estimator is biased. Because of the bias, the aforementioned (ϵ, δ)-approximation guarantee no longer holds for the truncated version of the tug-of-war sketch.

In Fig. 18, we plot the relative Rooted Mean Squared Error parameterized by several combinations of k and l. It shows that the sampling parameter l shifts the left wing of the relative error curve, while the truncation parameter k shifts the right wing.

4.5.3 Impact of bit errors on counters and protection

In this section, we discuss the types of error detection mechanisms that are appropriate for protecting our enhanced sketch and derive the formula for analyzing their error probabilities. An error detection code can be defined by its generating matrix. For example matrix $\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$ means there are two parity bits per counter. The first parity bit is XOR of all five bits and the second is the XOR of the three most significant bits. A counter is considered corrupted during transmission if it fails at least one of the parity checks. A corrupted counter thus detected will not be used in the BER estimation. However, all corrupted counters may not be detected. The following analysis will derive the distribution,

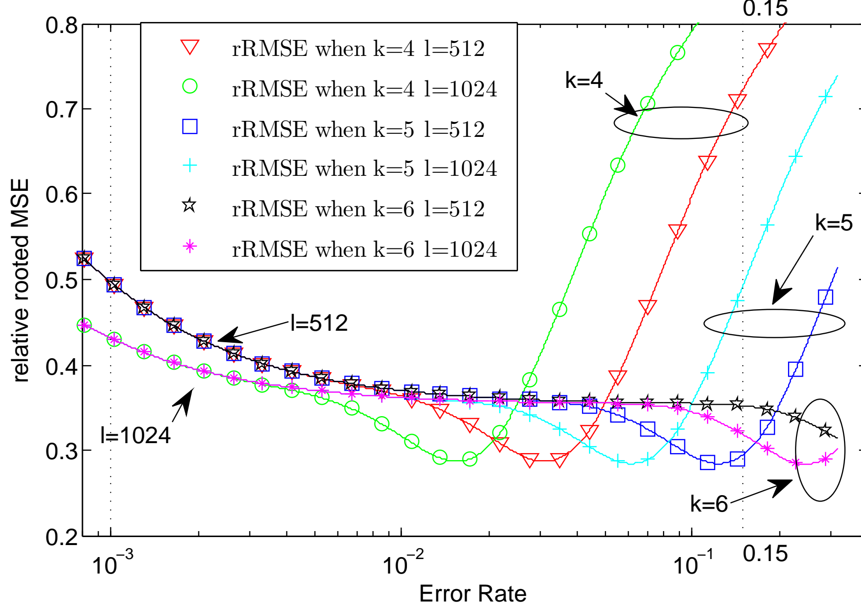


Figure 18: The relative Rooted Mean Squared Error of the enhanced tug-of-war sketch (fully protected) with different sampling and truncation parameters, when $c = 16, l = \{512, 1024\}, k = \{4, 5, 6\}$.

expectation and variance of the sketch differences Y, denoted by $\boldsymbol{\gamma_q(\theta)}$, when the effects of both types of corruptions (detected and undetected) are factored. Based on this analysis, our scheme chooses to have one parity bit per counter, which is the XOR of all 5 bits in the counter (corresponding to the first row of the above matrix).

We first model the distributions of the errors that survive the parity checking (undetectable errors). The impact of those errors on the estimates Y_i can be calculated as follows:

$$\boldsymbol{\gamma_q(\theta)} \approx \boldsymbol{\gamma}(\theta)\boldsymbol{Q}(\theta),$$

where $\boldsymbol{Q}(\theta)$ is determined by the design of the parity bit(s). We can then replace the $\boldsymbol{\gamma_q}$ in (30-32) with $\boldsymbol{\gamma_q(\theta)}$ to derive the expectation, variance and MSE of the final estimate Y_i^2.

The next step is to take into consideration the impact of detectable errors. Such errors will not affect the bias of the final $\hat{\theta}$, but will increase its MSE because fewer counters are

included in the average. The MSE of $\hat{\theta}$ when considering such errors is as follows:

$$
\begin{aligned}
l^2\mathbf{MSE}[\hat{\theta}] &= \sum_{k=1}^{c} \frac{1}{k}\binom{c}{k}\theta^k(1-\theta)^{c-k}\mathbf{Var}[Y^2] \\
&+ (\mathbf{E}[Y^2] - pl)^2,
\end{aligned} \tag{33}
$$

where θ is the probability with which each counter survives the parity checking. As discussed earlier, θ is a function of θ and the generating matrix. The error distribution of the final estimator is also numerically computable, but we omit the details here in the interest of space.

We compare different constructions of generating matrices in Fig. 19 and the results can be summarized as follows. First, it is not necessary to parity-check all sketch bits. However, if too few sketch bits are parity-checked, the accuracies in estimating low BER can be impaired. This phenomena can be observed by comparing the first three curves in the legend. Second, two parity bits per counter instead of one does not considerably improve the accuracy of the final estimation. For this reason, we choose to have one parity bit per counter in our scheme.

4.6 Fisher Information Analysis of EEC

In this section, we analyze the Fisher information contained in an EEC codeword. Since each bit in an EEC codeword is generated in the same way independent of each other, it suffices to analyze the contribution from each bit. The Fisher information of the codeword is simply the sum of the Fisher information contained in each bit.

Throughout this chapter, we will use notation $\check{z}_i$'s to denote the codeword bits (sent along with the packet) received (hence subjected to transmission errors), and use z_i''s to denote the codeword bits calculated from the packet received. In the EEC scheme, z_i and z_i' are the parity bits of the l_i bits on the same locations of the original and the received packets. The receiver computes their difference $X_i = \check{z}_i \oplus z_i'$, $i = 1, 2, \ldots, m$, and infers the error rate θ from the X_i's.

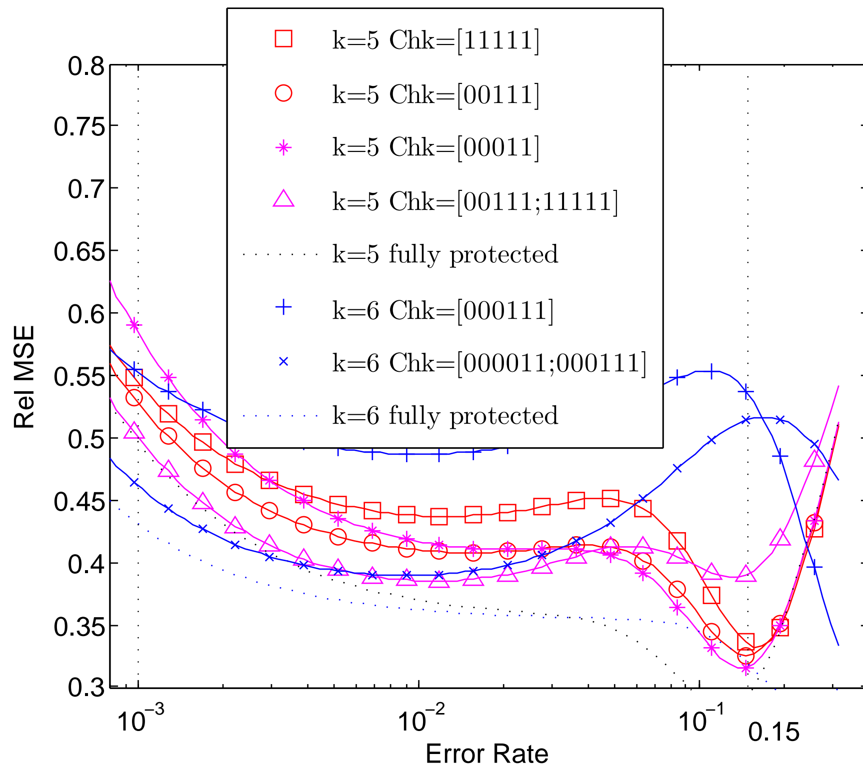


Figure 19: Comparisons of different constructions of parity-checking bits. When $k = 5$, we use sampling length $l = 512$; when $k = 6$, we use $l = 1024$

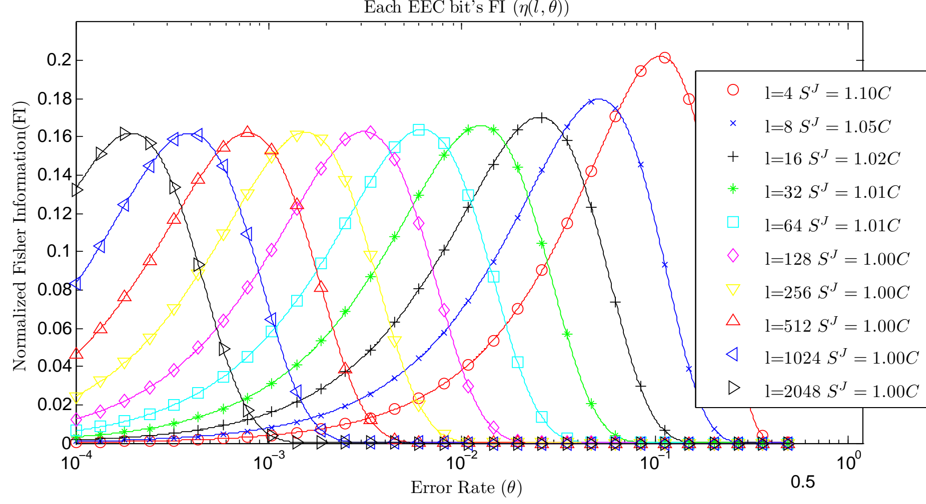


Figure 20: Fisher Information (of $\log\theta$) of each EEC bit in function of l and θ

4.6.1 Fisher Information Contained in each EEC bit

To simplify the exposition, we first perform the Fisher information analysis under the unrealistic assumption that the codeword is immune from corruption during transmission. In this case, $\check{z}_i$'s, the codeword bits received, are the same as z_i's, the codeword bits sent. We will then handle the more realistic case, without immunity, at the end of this section.

4.6.1.1 The case with "immunity"

Recall X_i defined above indicates whether or not the i_{th} parity equation holds. In the case with immunity, the likelihood function of observing $X_i = 1$ (i.e., odd number of bits "flipped" during transmission among the set of bits sampled) is as follows:

$$
\begin{aligned}
f(X_i = 1, \theta) &= \Pr(X_i = 1|\theta) && (34)\\
&= \sum_{j=0}^{2j+1\le l_i} \binom{l_i}{2j+1} \theta^{2j+1}(1-\theta)^{l_i-2j-1} \\
&= \frac{1-(1-2\theta)^{l_i}}{2}. && (35)
\end{aligned}
$$

The Fisher information contained in each X_i can be calculated as follows:

$$\begin{aligned} J_i^{EEC}(\theta) &= \mathbf{E}_\theta[(\frac{\partial}{\partial\theta}\log f(X_i;\theta))^2] && (36) \\ &= \Pr(X_i=1|\theta)(\frac{\partial}{\partial\theta}\log f(X_i=1;\theta))^2 + \Pr(X_i=0|\theta)(\frac{\partial}{\partial\theta}\log f(X_i=0;\theta))^2 \\ &= \frac{1-(1-2\theta)^{l_i}}{2}\left(\frac{2l_i(1-2\theta)^{l_i-1}}{1-(1-2\theta)^{l_i}}\right)^2 + \frac{1+(1-2\theta)^{l_i}}{2}\left(-\frac{2l_i(1-2\theta)^{l_i-1}}{1+(1-2\theta)^{l_i}}\right)^2 \\ &= \frac{4l_i^2(1-2\theta)^{2l_i-2}}{1-(1-2\theta)^{2l_i}}. && (37) \end{aligned}$$

Before we aggregate Fisher information contributed by all the codeword bits, we would like to highlight some nice properties of the derived Fisher information for a single bit (37). For convenience of comparisons, we define the Fisher information (of $\log\theta$) of each bit as $\eta(l_i,\theta)$:

$$\theta^2 J_i^{EEC}(\theta) = \frac{4\theta^2 l_i^2(1-2\theta)^{2l_i-2}}{1-(1-2\theta)^{2l_i}} \triangleq \eta(l_i,\theta). \quad (38)$$

As we explained earlier, this is inversely proportional to the Cramer-Rao bound of the relative variance. The larger $\eta(l,\theta)$, the tighter the bound of the relative error (and the log-difference) that can be achieved.

Values of $\eta(l_i,\theta)$ for various l_i (number of bits sampled) values are plotted in Figure 20. We can see that the $\eta(l,\theta)$ curves corresponding to different l values are actually very similar in shape to each other. A larger parity group size l is better for estimating smaller θ values and vice versa. These curves also have similar "heights" except when l gets really small (targeting extremely high BER close to the maximum possible value of 0.5). This means that the "peak estimation powers" of different parity bits are about the same. The maximum (i.e., "height") of each curve is always reached around $\theta = 0.4/l$, which means the parity bit computed from l sampled bits yields the best estimation for θ when θ is around $0.4/l$.

We can even quantify how much "information about θ" flows into the whole estimation spectrum by the following integral formula, which is the area covered by the FI(rel) curves in Figure 20:

$$\mathbf{S}^{\mathbf{J}}(l) = \int_0^{0.5} \theta^2 J(l,\theta)\, d\log\theta \quad (39)$$

In (39), the upper limit of the integral is 0.5 is because all formulas derived above (starting from (35)) are only valid when θ is less than or equal to 0.5.

From a practical perspective, an integral over the full spectrum might not be too useful since practical applications might be only interested in a particular range of spectrum rather than a full spectrum, such as $[10^{-3}, 0.15]$ proposed in [13]. Moreover, maintaining accuracy over a certain threshold might be even more practical than an integral. However, we find $\mathbf{S^J}$ is still a very good indicator of performance since it remains almost the same for EEC bits with different parameter l, which is not surprising since the Fisher information curves are similar to each other in shape and hence the "area under" the curves are also close to one another. In Figure 20, we list the values of $\mathbf{S^J}(l)$ of different l's in the legend, where constant $C = \frac{1}{24}\pi^2 \approx 0.4112$, which is a provable limit of $\mathbf{S^J}(l)$ when l goes infinity. When l increases, $\mathbf{S^J}(l)$ gets closer to C. In other words, the total information $\mathbf{S^J}$ contributed by each bit is all on the same scale and almost invariant to parameter l. In Section 4.7.4, we will see that this criteria $\mathbf{S^J}$ helps us to differentiate the strengths of different schemes and guides us in selecting better schemes.

4.6.1.2 The case without "immunity"

We proceed to perform the Fisher information analysis of EEC when the codeword sent along with the packet is no longer assumed to be immune from corruptions during transmission. The amount of Fisher information contained in each parity bit X_i can be derived as in (40). We omit the details of this derivation since it is a special case of the Fisher information analysis (without "immunity") in the gEEC framework in Section 4.7.2. Note that, unlike the case with "immunity" where the analysis rigorously holds, the analysis for the case without "immunity" only rigorously holds when i.i.d. random binary errors are assumed or random placement of bits are assumed.

$$\theta^2 J_i^{EEC}(\theta) = \frac{4(l_i+1)^2(1-2\theta)^{2l_i}}{1-(1-2\theta)^{2l_i+2}} \tag{40}$$

Notice that the RHS of (40) is equal to $\eta(l_i + 1, \theta)$, only slightly different from (37).

4.6.2 Combining the contributions of the different levels of bits

Since all parity bit are calculated from independent samples with replacement and hence each of them is independent of every other, the Fisher information of the codeword is simply the sum of their Fisher information together:

$$J^{EEC}(\theta) = \sum_{i=1}^{m} J_i^{EEC}(\theta). \quad (41)$$

In Figure 21, we plot the Cramer-Rao bound of the EEC scheme with the typical configuration (9 levels, 32 bits per level) and we also plot the Cramer-Rao bound of schemes with only one level of 32 bits. From Figure 21, we can see that after using the information of all levels, the estimation accuracy has the potential to be considerably improved. In other words, suppose there are two optimal estimators, the first can use only the information contained in one level of bits, while the second can use the information contained in all levels. Then the variance of the first estimator will be four times as large as compared to the second for most θ values. This means the codeword size has to be four times as large for the first estimator to match the second in estimation accuracy.

In the design of the original estimator in [13], they first identify the level of bits likely to be the most accurate for estimating θ and then only use that level of bits for estimation. Hence only one level of information is used in the final estimate. In Section 3.5 of the technical report [12], the authors have proposed an improved estimator that can make use of two neighboring levels of parity bits. In Figure 21, we use $\hat{\theta}_1$ to denote the original one proposed in [13] and use $\hat{\theta}_2$ to denote the improved version proposed in [12]. We can see that neither estimator is close to the Cramer-Rao bound corresponding to the amount of Fisher information contained in such one or two levels of bits (In other words, their estimators have not made full use of even the information contained in such one or two levels of bits) and far from the Cramer-Rao bound corresponding to the Fisher information of all the codeword bits. We note that when θ is larger than 0.15, the relative MSE falls

below the Cramer-Rao bound; This is because the estimator is actually very biased in that region. According to (17), this might lead to smaller MSE.

4.6.3 MLE estimator for EEC scheme

In this section, we present our MLE decoder that matches the Cramer-Rao bound, shown in the following formula.

$$\begin{aligned} \hat{\theta}_{MLE} &= \arg\max_{\theta}\{\sum_{i=1}^{m} \log \Pr(X_i|\theta)\} && (42) \\ &= \arg\max_{\theta}\{\sum_{i=1}^{m} \log \frac{1-(1-2\theta)^{l_i}}{1+(1-2\theta)^{l_i}} X_i && (43) \\ &\quad + \log(1+(1-2\theta)^{l_i})\} && (44) \end{aligned}$$

It has been discovered in [24] that the non-informative prior, Jefferys invariant prior, can remove the $O(\frac{1}{N})$ component in bias for the family of exponential models. Here, although the likelihood function is not a closed-form distribution, we find that Jeffreys prior will help the MLE estimator to achieve better results. The MLE with the Jefferys prior, denoted as $\hat{\theta}_{MLE_j}$, is

$$\begin{aligned} \hat{\theta}_{MLE_j} &= \arg\max_{\theta}\{\sum_{i=1}^{m} \log \Pr(X_i|\theta) + \frac{1}{2}\log J(\theta)\} \\ &= \arg\max_{\theta}\{\sum_{i=1}^{m} \log \frac{1-(1-2\theta)^{l_i}}{1+(1-2\theta)^{l_i}} (X_i - \frac{1}{2}) \\ &\quad + \log(2l_i(1-2\theta)^{l_i-1})\}. \end{aligned}$$

Here $p(\theta) \sim \sqrt{J(\theta)}$, where $p(\theta)$ is the a priori distribution of the parameter θ and $J(\theta)$ is the Fisher information calculated in (41). Note that the MLE with Jefferys prior will still asymptotically reach the Cramer-Rao bound.

4.7 Design and Fisher Information Analysis of Generalized EEC (gEEC)

In the previous section, we showed through Fisher information analysis that the original decoder for EEC used in [13] is far from optimal and our new decoder is near-optimal (by

almost matching the Cramer-Rao bound) given the amount of Fisher information contained in an EEC codeword. Now we would like to find out whether the EEC encoding scheme is efficient enough by comparing its Fisher information (per bit) with that of EToW. However, when we were performing the Fisher information analysis of EToW, we discovered a generalized EEC (gEEC) scheme that can be parameterized into both EEC and EToW. Fisher information analysis of EToW can thus be generalized to that of gEEC so we shift our "target" for comparison to gEEC. Through this unified framework of gEEC, we have found that some parameterizations of the gEEC family contain 25% or more Fisher information per bit in their codewords than EEC. In other words, the EEC encoding scheme is not very efficient either. The discovery of gEEC is important also for another reason: We have discovered a unified decoder (estimator) for gEEC that is near-optimal (by almost matching the Cramer-Rao bound) and when parameterized into EToW, is more accurate than the original EToW decoder proposed in [30]. Strictly speaking, it is not the clear winner however since the original EToW decoder has much lower computational and storage complexities.

The rest of the section is organized as follows. In Section 4.7.1, we briefly describe the gEEC encoding scheme by highlighting its differences from and connections to both EToW and EEC. In Section 4.7.2, we proceed to perform the Fisher information analysis of its codeword and design its estimator. We will see the gEEC decoder that almost matches the Cramer-Rao bound in Section 4.8. Finally, we will present the numerical results of the information contained in gEEC family in Section 4.7.4.

4.7.1 The gEEC Encoding

A gEEC codeword (called *sketch*) of a packet (viewed as binary vector $\vec{b}$) consists of m sub-sketches $z_1, z_2, \ldots, z_m$. The value of sub-sketch z_i is set to the number of 1's contained in the binary vector generated by sampling l_i bits from the packet *with replacement*, which we refer to as $\vec{b_i}$, and then XOR-ing it with a pseudo random vector $\vec{s_i}$ bitwise. This operation is

the same as in EToW, except that in EToW the value of z_i is set to the half of the difference between the number of 1's and the number of 0's. It can be shown that these two types of encodings can be made equally efficient and statistically equivalent with respect to the truncation operations (described shortly) with proper parameterizations.

The main difference between EToW and gEEC is that all $l_i's$ have to take the same value in EToW. This is not an artificially crafted difference because EToW's decoder, inherited from ToW and based on the method of moments, imposes this equal length requirement, while the new decoder we propose for gEEC that we will describe in Section 4.7.2.2, does not have such a requirement due to its MLE nature. Like in EToW, we use barely enough bits to encode each sub-sketch and "overflows" are handled in the same way through truncation. Moreover, different from EToW which needs extra checking bits to detect corruption inside the sketch, we will see that the estimators provided the gEEC's framework have built-in capability to decode the sketches which might suffer corruption in the case without "immunity".

The precise definition of z_i in gEEC scheme is as follows:

$$z_i = \sum_{j=1}^{l_i} (b_{i,j} \oplus \frac{1 + s_{i,j}}{2}) \qquad (\text{mod } K_i), \tag{45}$$

where $K_i = 2^{k_i}$, k_i is the number of bits allocated to sub-sketch z_i. Each $s_{i,j}$ is a pre-computed pseudo-random number uniformly and independently selected from $\{-1, 1\}$, the same as the definition in the tug-of-war sketch [3, 30]. The function $\frac{1+s_{i,j}}{2}$ maps $s_{i,j}$ from $\{-1, 1\}$ to $\{0, 1\}$.

Noticed that here, different from the definition in (23) of Section 4.5,here we directly define the z_i by xor sum and z_i is no longer defined in $\{-2^{k_i-1}, -2^{k_i-1} + 1, ..., 2^{k_i-1} - 1\}$, but defined in $\{0, 1, ..., 2^{k_i} - 1\}$. We should say the purpose of this is only to make the formulas cleaner and is also to be consistent with the definition in [31].

As shown below, the nature of the definition above is a random projection of $\vec{s_i}$ (only different from the $\vec{b_i} \cdot \vec{s_i}$ by a pseudo-random constant, i.e. a number that is the same in both

sender and receiver):

$$z_i = \sum_{j=1}^{l_i} (b_{i,j} \oplus \frac{1 + s_{i,j}}{2}) \quad (46)$$

$$= \sum_{j=1}^{l_i} \frac{1 + (2b_{i,j} - 1)s_{i,j}}{2} \quad (47)$$

$$= \vec{b_i} \cdot \vec{s_i} - \frac{1}{2}\vec{1} \cdot \vec{s_i} + \frac{l_i}{2}. \quad (48)$$

It can be shown that when we allocate only 1 bit for each sub-sketch, the sketch becomes a parity array and in this case gEEC "degenerates" into a scheme statistically equivalent to EEC with the same l_i values. We can also see that gEEC becomes statistically equivalent to EToW without sketch protection (explained next) when $l_i's$ are set to the same value.

4.7.2 Fisher information analysis

In this section, we analyze the Fisher information contained in each gEEC codeword. Recall from Sec. 4.4.1 that in EToW we need to protect the sketch against corruptions during transmission using lightweight error-detection codes. We will show no such protection is needed in gEEC because, unlike the EToW decoder, the proposed MLE decoder for gEEC is robust against such corruptions. In order to simplify the presentation of the analysis, we first analyze the Fisher information of a gEEC codeword assuming that the codeword is immune from corruption during transmission in Section 4.7.2.1 and then show how to remove this assumption in Section 4.7.2.2.

4.7.2.1 The case with "immunity"

Like in Section 4.6, we use $\vec{b_i'}$ to denote the set of bits sampled *with replacement* from the packet received (subject to corruptions during transmission) that are used to compute the i_{th} sub-sketch z'_i at the receiver side, and use $\vec{b_i}$ to denote the corresponding set of bits sampled from the packet sent (can be different from $\vec{b_i'}$ due to corruptions during transmission) that are used to compute the i_{th} sub-sketch z_i at the receiver side. In this section, we derive

the Fisher information of the sketch under the aforementioned immunity assumption that these sub-sketches z_1, z_2, ..., z_m will arrive at the receiver without having any of their bits corrupted during transmission.

The receiver can calculate the difference X_i between z'_i and z_i

$$X_i \triangleq z'_i - z_i (\text{mod } K), i \in [m] \tag{49}$$

Here $K = 2^k$ where k is the number of bits we allocate to each sub-sketch. The effect of the aforementioned (possible) overflow and resulting truncation is reflected in "modulo K". It can be shown that this observation X_i is the following function of the error vector $\vec{e_i}$ (the difference vector between $\vec{b_i}$ and $\vec{b'_i}$), where s_i is the aforementioned pseudorandom vector over which the sampled bit vectors $\vec{b_i}$ and $\vec{b'_i}$ are linearly projected:

$$X_i = \vec{b_i} \cdot \vec{s_i} - \vec{b'}_i \cdot \vec{s_i} (\text{mod } K), \text{due to (48)} \tag{50}$$

$$= \vec{e_i} \cdot \vec{s_i} (\text{mod } K). \tag{51}$$

Since observations X_1, X_2, ..., X_m are independent random variables, the likelihood function of the random vector $\langle X_1, X_2, ..., X_m \rangle$ is the product of the likelihood functions of these random variables. The likelihood function of X_i (for arbitrary i) can be derived as follows.

For convenience, we drop the subscript from X_i and denote it simply as X. In the following we will derive the probability mass function (PMF) of X, which takes values from the set of $K = 2^k$ integers $\{0, 1, ..., K-1\}$. It can be shown that its PMF can be determined by a K-dimensional vector $\vec{\gamma}(\theta, l) \equiv \langle \gamma_0(\theta, l), \gamma_1(\theta, l), ..., \gamma_K(\theta, l) \rangle$ where $\gamma_i(\theta, l) \equiv \Pr(X = i|\theta, l)$, $i \in [0, 1, ..., K-1]$. Note that each scalar γ_i is a function of the error rate θ and the number of bits sampled l (here the subscript i is dropped from l_i). We show that $\vec{\gamma}(\theta, l)$ can be computed from the following recurrence relation.

Lemma 5.

$$\vec{\gamma}(\theta, l)_{1\times K} = \vec{\gamma}(\theta, l-1)_{1\times K} \mathbf{M}(\theta)_{K\times K}, \tag{52}$$

,where $\mathbf{M}(\theta)$ *is the same as the* $\mathbf{M}(\theta)$ *defined in (26) of Section 4.5.*

$$\mathbf{M}(\theta) = \begin{bmatrix} \textit{1-}\theta & \theta/2 & \cdots 0 \cdots & & & \theta/2 \\ \theta/2 & \textit{1-}\theta & \theta/2 & \cdots 0 \cdots & & \\ & \theta/2 & \textit{1-}\theta & \theta/2 & \cdots 0 \cdots & \\ & \cdots & & \cdots & & \\ & \cdots 0 \cdots & & \theta/2 & \textit{1-}\theta & \theta/2 \\ \theta/2 & & \cdots 0 \cdots & & \theta/2 & \textit{1-}\theta \end{bmatrix}_{K\times K} \tag{53}$$

The only difference from the definitions of Section 4.5 is in the initial condition of gam⃗ma, which is

$$\vec{\gamma}(\theta, 0) = [1, 0 \cdots, 0]_{1\times K}. \tag{54}$$

here.

Proof. Let $\vec{e} = \langle e_1, e_2, \cdots, e_l\rangle$ and $\vec{s} = \langle s_1, s_2, \cdots, s_l\rangle$. We define the following interim random variables Y_j, $j = 0, \cdots, l$:

$$Y_j = \sum_{k=1}^{j} e_k s_k. \tag{55}$$

Clearly, $X = Y_l (\text{mod } K)$. Due to the sampling with replacement policy, the increment in each step $\Delta Y_j = Y_{j+1} - Y_j = e_{j+1}s_{j+1}$ is independent of every other. Hence the random variables $\{Y_j\}_{0\le j\le l}$ make up a Markov chain. In each step, with probability $1-\theta$ an unchanged bit is selected and hence $\Delta Y_j = 0$; with probability θ a changed bit is selected, and half of these increments are $+1$ and the other half -1 since s_k is uniformly at random from $\{-1, 1\}$.

Hence the distribution of ΔY_j is:

$$\Delta Y_j = \begin{cases} -1 & \text{with prob } \theta/2, \\ 0 & \text{with prob } 1-\theta, \\ 1 & \text{with prob } \theta/2. \end{cases} \tag{56}$$

Mapping $\{Y_j\}_{0\le j\le l}$ into the finite field Z_K, formula (56) becomes the transition matrix, which is the circular matrix M defined in (53). □

To allow for efficient matrix computation, $\mathbf{M}(\theta)$ can be diagonalized as follows.

$$\mathbf{M}(\theta) = \frac{1}{K}\mathbf{\Omega}' Diag(d_0, d_1, \ldots, d_{K-1})\mathbf{\Omega} \tag{57}$$

where $\Omega = \{\omega_{ik}\}$, $\omega_{ik} = \exp(\frac{2\pi ikj}{K})$, j is the imaginary unit, and $d_i = 1 - \alpha_i\theta$, $\alpha_i = 2\sin^2(i\pi/K)$, which is actually the Fourier transform matrix.

Considering that $\vec{\gamma}(\theta, 0) = [1, 0 \cdots, 0]_{1\times K}$, we have

$$\begin{aligned}
\vec{\gamma}(\theta, l) &= [1, 0, \cdots, 0]\mathbf{M}(\theta)^l \\
&= \frac{1}{K}[1, 0, \cdots, 0]\mathbf{\Omega}' Diag(d_0^l, d_1^l, \ldots, d_{K-1}^l)\mathbf{\Omega} \\
&= \frac{1}{K}[d_0^l, d_1^l, \ldots, d_{K-1}^l]\mathbf{\Omega}.
\end{aligned} \tag{58}$$

The Fisher information of the gEEC sketch can be calculated as follows:

$$\begin{aligned}
J(\theta, l) &= E_\theta(\frac{\partial}{\partial\theta}\log f(X_i; \theta))^2 \\
&= \sum_{j=0}^{K-1}\gamma_j(\theta)(\frac{\mathrm{d}}{\mathrm{d}\theta}\log(\gamma_j(\theta)))^2 \\
&= \sum_{j=0}^{K-1}\frac{l^2\{[0, \alpha_1 d_1^{l-1}, \ldots, \alpha_K d_{K-1}^{l-1}]\mathbf{\Omega}\}_j^2}{K\{[d_0^l, d_1^l, \ldots, d_{K-1}^l]\mathbf{\Omega}\}_j},
\end{aligned} \tag{59}$$

where γ_j denotes the jth item in vector $\vec{\gamma}$ and $\{\vec{v}\}_j$ denotes the jth item in vector $\vec{v}$.

It can be shown that when we set k to 1 (so that gEEC degenerates into EEC), formula (59), the Fisher information of gEEC codeword is equal to formula (37), that of EEC.

4.7.2.2 The case without immunity

In the previous section, we have performed an Fisher information analysis of gEEC under the assumption that the codewords (sketches) sent along with the packets are not subject to corruptions during transmission (i.e., "with immunity"). In reality, these codewords are certainly not immune to bit errors. In this section, we perform the Fisher information anaysis without this "immunity assumption".

Suppose the sender sends out a sub-sketch z_i and the receiver receives $\check{z}_i$. Now $\check{z}_i$ may differ from z_i as the "immunity" has been taken away. Having no knowledge of z_i, the

receiver has to use $\check{z}_i$ and z_i' computed from the received packet to infer the bit error rate θ. The conditional (upon θ) joint probability mass function of $\langle \check{z}_i, z_i' \rangle$ is shown as follows. For ease of notation, we remove the subscript i from the notations and get:

$$\begin{aligned}
\Pr(\check{z}, z'|\theta) &\propto \sum_{z=0}^{K-1} \Pr(z, \check{z}|\theta) \Pr(z'|\theta, z) \\
&\propto \sum_{z=0}^{K-1} \Pr(z) \Pr(\check{z}|\theta, z) \Pr(z'|\theta, z).
\end{aligned} \tag{60}$$

Note that the formula (60) above only holds under the assumption that the binary error flips on sketch bits and the error flips on data bits are independent, i.e., $\check{z} \perp z'|\theta, z$, which requires either that the binary errors are i.i.d. distributed inside the sketch or that the locations of all bits participating including the sketch bits are all sampled with replacement from the packet, which is not possible to be strictly guaranteed in reality. Hence, we readily admit that, different from the analysis for the case with "immunity" which is rigorously held, the analysis for the case without "immunity" does not model reality perfectly, though we believe that it should be very close.

The matrix representation of (60) is as follows:

$$\mathbf{P}(\theta)^{final}_{K\times K} \propto \mathbf{\Lambda}\mathbf{M}(\theta)^{l}\mathbf{T}(\theta)_{K\times K}, \tag{61}$$

In (61), matrix $\mathbf{M}$ corresponds to $\Pr(z'|\theta, z)$ and can be calculated using (57). Matrix $\mathbf{\Lambda}$ is a diagonal matrix and corresponds to $\Pr(z)$. Its diagonal elements can also be calculated using (58) with $\theta = 1$. We acknowledge that an ulterior motive for us to define z_i as the number of 1's, rather than the number of 1's minus the number of 0's, in the bitwise-XOR of $\vec{b_i}$ and $\vec{s_i}$, is that it makes these formulae much "cleaner".

Matrix $\mathbf{T}$ is the transition matrix that corresponds to $\Pr(\check{z}, \check{q}|\theta, z)$. Each of its entry T_{ij} is defined as follows:

$$\mathbf{T}_{i,j} = \theta^{d_{ij}}(1-\theta)^{k-d_{ij}}, \tag{62}$$

where d_{ij} is the Hamming distance between the binary representation of i and j.

All entries in (61) are differentiable and the Fisher information can be derived in a way similar to (59). In the interest of space the final Fisher information formula is omitted here.

Note that when $k = 1$, gEEC degenerates to EEC, and the final Fisher information formula can be shown to be equivalent to EEC's Fisher information formula (40).

4.7.3 Our MLE Estimators

In this section, we derive the MLE decoder for gEEC that perform much better than both EEC and EToW decoders, as we will show in Sec. 4.8. Again, we first derive it for the easier case with "immunity" and then proceed to take the "immunity" away.

In the case with "immunity", our observations are X_i, which is the difference between each pair of $\check{z}_i$ and z'_i, $i = 1, 2, ..., m$, and our goal is to estimate θ. The maximum likelihood estimator is

$$\hat{\theta}_{MLE} = \arg\max_{\theta}\{\sum_{i=1}^{m} \log\{\vec{\gamma}(\theta, l_i)\}_{X_i}\} \tag{63}$$

Here $\{\vec{\gamma}(\theta, l_i)\}_{X_i}$ denotes the X_i^{th} scalar in $\vec{\gamma}$ and $\vec{\gamma}(\theta, l)$ and $J(\theta, l)$ have both been derived earlier. The MLE estimator with Jeffreys prior (MLE-J) is

$$\hat{\theta}_{MLE-J} = \arg\max_{\theta}\{\frac{1}{2}\log J(\theta) + \sum_{i=1}^{m} \log\{\vec{\gamma}(\theta, l_i)\}_{X_i}\} \tag{64}$$

We introduce $MLE - J$ because it performs better than the plain vanilla MLE empirically, which will show in Section 4.8.

Similar to (64), the MLE estimator with Jeffreys prior for the case without immunity is

$$\hat{\theta} = \arg\max_{\theta}\left\{\frac{1}{2}\log J(\theta) + \sum_{i=1}^{m} \log\{\mathbf{P}(\theta, l_i)\}_{\check{z}_i, z'_i}\right\}, \tag{65}$$

where $\{\mathbf{P}(\theta, l_i)\}_{\check{z}_i, z'_i}$ denotes the $(\check{z}_i, z'_i)^{\text{th}}$ entry in the probability matrix $\mathbf{P}$.

This estimator will also be evaluated empirically in Section 4.8.

4.7.4 Numerical Results of the Fisher Information contained in the gEEC Family

Fisher information contained in each gEEC's sub-sketch is determined by two factors: the sampling group size l and the binary width k. For convenience, this is denoted as

gEEC(l, k), and a sketch composed of m such sub-sketches is denoted by gEEC(m, l, k).

In Figure 22, we show the impact of different l's and k's in the case with immunity. We observe that, similar to Figure 20, the parameter l only "shifts" the curve. The larger l, the more resolution on lower θ's, while the total amount of "information flow" S^J remains almost the same. The parameter k, on the other hand, expands the "span" of the curve, leading to a wider spread of the estimation power on the spectrum.

To compare the total amount of "information flow" of different parameterizations more conveniently, we also list the value S^J—the area covered by each curve as defined in (39)—of each parameterization in the legend of the figure, where the constant C is $\frac{\pi^2}{24} \approx 0.4112$, the lower limit of the area covered by the EEC bit's Fisher information curve, as discussed in Section 4.6. Based on the S^J values in Figure 22, we conclude that $k = 2$ is not able to bring any additional benefit. In contrast, one sub-sketch with $k = 4$ can gain more information over 4 independently coded EEC bits.

The next question we ask is: How much information is lost due to the contamination of the sketch? To measure this, we plot the Fisher information curve with $k = 4$ or 6 and $l = 2048$, 512 or 128 in Figure 23. We observe that the Fisher information curve is impacted only when θ is not small and the loss of information is also modest. For some cases, such as $k = 6$ and $l = 128$, the total information is improved even when the sketch is subject to errors.

We summarize the impact of l and k's on S^J by listing the $\frac{1}{kC}S^J$ values of different parameterizations in Table 14. From the perspective of total information gain S^J, more bits per sub-sketch usually improves the gain. We see that $k = 5$ gives around 25% more information per bit in the contaminated case, which can be translated to a similar ratio of reduction of the sketch size to achieve the same variance bound. It can be also shown that the performance of one sub-sketch with $k = 5$ can dominate six one-bit sub-sketches together.

However, a larger k is not always better. The larger k gets, the wider the span of the

	Case with immunity			Case without immunity		
k	l=128	l=512	l=2048	l=128	l=512	l=2048
2	1.00	1.00	1.00	1.00	1.00	1.00
3	1.12	1.12	1.11	1.08	1.10	1.11
4	1.23	1.22	1.22	1.13	1.18	1.21
5	1.24	1.31	1.31	1.16	1.20	1.26
6	1.05	1.30	1.37	1.19	1.21	1.27
7	0.90	1.14	1.36	1.22	1.24	1.27

Table 14: The total information gain $\frac{S^J}{kC}$ ($C = \frac{1}{24}\pi^2$)

resolution curve gets, which might cover more range than needed. If l is small (for better resolution on large θ's), k should not need to be too large, otherwise it will be wasteful (such as the $k = 6$ and $l = 128$ case in the table). For a typical EEC application in wireless communications where the primary target parameter range is $[10^{-3}, 0.15]$, the span of $k = 5$ (or 6) would be sufficient. Moreover, a large k will mean a higher implementation cost for a relatively modest gain, which will be discussed soon.

4.8 Evaluation

4.8.1 Experimental Results

In this section, we evaluate the performance of our sketches experimentally and compare it with the original EEC scheme.

For the EEC scheme, we use the parameters recommended in [13] for Wifi applications, i.e., with 9 levels and each level comprised of 32 bits. In total the EEC scheme costs 288 bits per packet and is targeted for estimating error rates in the $[10^{-3}, 0.15]$ region. The authors of [13] have proposed three different estimators for their scheme. A naive estimator for $\hat{\theta}$ (BER) is $q_i/2^i$ (defined in their paper); Two more sophisticated and accurate estimators are the roots of $\phi(2^i, \theta) = q_i$ and $\phi(2^i, \theta) = q_i/2 + q_{i-1}(1 - q_{i-1})$ respectively. We find that the latter two estimators both have better estimation accuracies than the naive one, and neither of them dominates the other. For convenience, we use $\hat{\theta}_1$ and $\hat{\theta}_2$ to denote the latter estimators, respectively.

We have actually proposed two approaches. The first is the enhanced tug-of-war (EToW) sketch, as presented in Section 4.5. The second is the estimator and schemes in the gEEC framework, as presented in Section 4.7. We emphasize that although EToW can be regarded one special case in the gEEC framework, it uses a very different estimator with some pros and cons and hence should be evaluated separately. For both two approaches above, we aim to answer two questions: (1) How far is the estimator's real performance from the theoretical results? and (2) How does this compare with previously proposed solutions? Each value in all figures in this section is obtained with 1000 runs in our simulations, in the case without immunity.

As for the EToW sketch, as analyzed in Section 4.5, the sampling parameter l and truncation parameter k can be tuned for different target error rate regions. Since we will show that the experimental results are nearly identical to the analytical results, we present only the analytical results with two parameter settings: c=16 or 48, l=768, k=5, and one parity checking bit per counter generated by the matrix $\begin{bmatrix} 0 & 0 & 1 & 1 & 1 \end{bmatrix}$. The sketch with 16 counters consumes only 96 bits per sketch, 33.3% of that consumed by the original EEC scheme; The sketch with 48 counters consumes 288 bits, the same as the original EEC scheme.

As for gEEC approach, since we will also show that the experimental results are nearly identical to the analytical results, we present only the analytical results with the following parameterizations: the original EEC scheme with the new estimator(288bits), gEEC(16,512,5) (80bits), gEEC(16,768,6) (96bits),gEEC(56,512,5)(280bits) and gEEC(48,768,6)(288bits). We will also compare the performance of estimators with or without the Jeffrey's prior.

As for the performance metric used for comparisons, the comparison metrics that we use are the relative mean squared error (rMSE, defined as $\frac{1}{\theta}\overline{(\hat{\theta}-\theta)^2}$), the mean squared error of $\log\hat{\theta}$ (defined as $\overline{(\log\hat{\theta}-\log\theta)^2}$)), the ratio of large errors (the ratio of $\hat{\theta}$ that are larger than 2θ or smaller than $\theta/2$), and the relative bias ($\overline{\frac{\hat{\theta}}{\theta}-1}$).As discussed in Section 4.2.3, although the Cramer-Rao lower bound for the relative MSE of $\hat{\theta}$ (18) and the MSE of $\log\hat{\theta}$

(19) are the same, we prefer to use the statistics of $\log \hat{\theta}$ for comparison since it allocates larger penalty to large deviations. We will also compare the tail probabilities of different estimator on some θ's on some representative BER values, such as 0.005 and 0.05.

We will firstly go through the evaluation of EToW sketch in this subsection. One distinguished feature of the EToW sketch is that its performance is fully predictable.

The experimental results of EToW sketches are shown in Figure 4.8.1. Each curve is generated from the results obtained from 8000 experiments. We can make two observations from these results. First, we observe that all experimental results are nearly identical to the analytical results. Second, we observe that the performance (i.e., estimation accuracy) of the enhanced tug-of-war sketch of size 96 bits is close to or even better than the original EEC scheme of size 288 bits in the target error rate region, while the enhanced tug-of-war sketch of size 288 bits performs way better than the original EEC scheme. To summarize, our scheme achieves similar BER estimation accuracies with a sketch size that is only 1/3 of that used by the original EEC scheme.

In the next step, we will compare the performance of gEEC sketches and EToW sketch together.

In Figure 25, we compare the performance of the original 288-bit EEC scheme with original estimators with three gEEC/EToW configurations, gEEC$(16, 768, 5)$, gEEC$(16, 768, 6)$ and EToW$(16, 768, 5, r = 1)$, whose total transmission costs are 80, 96 and 96 bits, respectively. We see that our new sketches perform very well in $[0.001, 0.15]$ with much less transmission overhead than the original EEC, while the performance of the original EEC's estimator varies, especially when measured by harsher criteria such as the MSE of $\log \hat{\theta}$ and the ratio of large errors.

In Figure 26, we compare the performance of the four estimators of two schemes, the original EEC and gEEC$(56, 512, 5)$. All four estimators are derived from the newly proposed gEEC framework, two of which use the Jeffreys prior and the other two do not. We can see that the estimators with Jeffreys prior are generally better in the range where θ is

relatively large and the inherent resolution of the scheme is relatively weak, no matter if measured by MSE or by bias. Since estimators with Jeffereys prior are usually empirically better, we always use that version in other comparisons.

In Figure 27, we compare the performance of four schemes, the original EEC, gEEC$(56,512,5)$, gEEC$(56,768,6)$ and EToW$(56,768)$ with 5-bits per sub-sketch and 1-bit for detecting corruption, the transmission cost of which are almost the same. The first three use gEEC's MLE estimator with Jeffreys prior, while the last one uses EToW's moment-based estimator. Comparing these figures with Figure 25, we can see that, with the same transmission cost, the estimation accuracy is substantially improved. Moreover, we can see that the gEEC$(56,512,5)$ and gEEC$(56,768,6)$'s performance are generally better than the others. In a wide range they can achieve around a 30-50% reduction of MSE, compared with the EEC scheme with our new estimator. This is not surprising since we have already seen that a larger k can bring a modest improvement of estimation accuracy. We also observe that EToW's performance is only slightly worse, except when θ is close to or larger than 0.1.

Comparing the curves in Figure 25 and Figure 26, we can see that when the total number of sketches is large, the MLE estimator's performance is nearly equal to the Cramer-Rao lower bound, and its bias is also reduced as seen by comparing Figure 25(d) and Figure 25(d). Note that sometime the MSE of $\log\theta$ or rMSE might be below the Cramer-Rao bound when bias is high, which is possible since all curves of Cramer-Rao bound presented in the figures are for the unbiased estimator.

To summarize our comparisons above, our estimators, especially the ones with Jeffreys prior, can almost achieve the Cramer-Rao bound empirically. On one hand, gEEC$(16,768,5)$ (which is also EToW's scheme, just without EToW's extra error detection bit) can achieve a similar level of performance as the original EEC scheme with only one-fourth of the sketch size; on the other hand, with the same budget of transmission cost, the estimation accuracy of the original EEC scheme (288 bit design) can be greatly improved by our new estimators, and our gEEC design can achieve around 30% additional gain of estimation accuracy.

Compared to the EToW's performance, the gEEC's performance is still better, especially in the range that θ is relatively large.

4.8.2 Implementation Cost of Estimators and Selection of Parameters

All presentations thus far have focused on the estimation performance. However, the implementation and the computation costs should also be considered.

Firstly, the the implementation of EToW's estimator is very low, since it only requires a combination of inner products as well as modulo and other arithmetical operations, all of which have cost $O(n)$. In practice, the inner product is equivalent to the bit counting operation. All of these operation took very little time in our experiments and therefore we do not present any computation time measurements here.

As for the gEEC's estimator, which is an MLE estimator in nature, it can be implemented as table-based lookup. Whether or not it is enhanced by the Jeffrey's prior, the MLE estimator can be transformed in this way:

$$\hat{\theta}_{MLE} = \arg\max_{\theta}\{\mathrm{A}(\theta)\mathrm{Y} + \mathrm{B}(\theta)\}, \tag{66}$$

where each entry of Y indicates the count of one particular type of sub-sketch equals to one particular value. $\mathrm{A}(\theta)$ and $\mathrm{B}(\theta)$ are determined by

We can implement (66) as a linear transform of $\mathbf{Y}$, i.e. $\mathbf{A}_{d_1\times d_2}\mathbf{Y}_{d_2\times 1} + \mathbf{B}_{d_1\times 1}$ and then find the maximum of the result. Here $\mathbf{A}_{d_1\times d_2}$ and $\mathbf{B}_{d_1\times 1}$ are both pre-calculated matrixes.

One of $\mathbf{A}$'s dimensions, d_1, is determined by the size of candidate θ's. For most practical EEC applications, d_1 in the order of hundreds is sufficient since it will be wasteful if it becomes more fine-grained than the spectrum of estimation.

$\mathbf{A}$'s other dimension, d_2, corresponding to the length of $\mathbf{Y}$, depends on the design of the sketch. Suppose the codeword is composed of c types of sub-sketches: m_1 gEEC(l_1, k_1) sub-sketches, m_2 gEEC(l_2, k_2) sub-sketches, $\ldots, m_c$ gEEC(l_c, k_c) sub-sketches. In the case of "immunity" where the estimator can directly infer from the difference between $\check{z}_i$ and z'_i, d_2 equals to $\sum_{i=1}^{c}(2^{k_i} - 1)$. In the case without "immunity," d_2 equals to $\sum_{i=1}^{c}(2^{2k_i} - 1)$,

since in such a case the estimator should directly infer from the pair of $\check{z}_i$ and z'_i and hence the table size is squared.

On one hand, the MLE estimator above can be implemented with an extremely low cost, when all sub-sketches' k parameters equal 1, which means the scheme degenerates to the original EEC scheme. In this case, the matrix $\mathbf{A}$ can be as small as 9×100 (say, d_2=100). Moreover, some iterative methods similar to the bisection method can be employed to further reduce the number of multiplication, add and compare operations to about 90. Furthermore, since our estimator has strong built-in capability to combine the information from different levels, it can be shown that a 3-level, 96-bit-per-level design performs very closely to a 9-level 32-bit design, which can reduce the number of the sub-sketch types and hence even further reduce the cost by two-thirds.

On the other hand, as shown by previous evaluation results, larger values of k in the sub-sketch, such as 5 or 6, can generally bring around 25% of additional improvement of estimation accuracy, at the cost of thousands times higher cost in the storage and the lookups in tables $\mathbf{A}$ and $\mathbf{B}$. The computation cost actually remains almost the same since $\mathbf{X}$ is sparse, and we can implement the linear transform by summing up only few rows. Hence, the applicability of this improvement depends on the application scenario, since a lookup table of several hundreds KB might be a very small cost for some applications, but infeasible for others. Also, note that as discussed in Sec. 4.7.4, a large k might not be necessary if the target range of parameters is not so wide.

There are two middle paths between the cases above. One way is to use the combinations of several sub-sketches with $k = 3$ or 4 and different l's, the cost of which is much smaller than $k = 5$ since the table size increases exponentially ($O(2^{2k})$) in the case without immunity, while the scheme can still receive some gain on estimation accuracy.

Another way is to use the EToW's scheme and estimator, whose implementation cost is much lower. The weakness is that the performance is much weaker, especially when the error rate θ is large.

In summary, the gEEC framework (including the EToW sketch) can be easily and flexibly configured for different requirements of estimation accuracy. All guidelines discussed in this section not only hold for θ in the range $[0.001, 0.15]$, but also for the other arbitrary ranges as needed.

4.9 Conclusions

The seminal work of Chen *et al.* [13] has opened the door for the design of high-quality error estimating codes, with applications towards improving wireless network performance. Chen *et al.* [13] designed an exceedingly simple code for estimating bit error rates in packets being transmitted, and it is an open—yet challenging—question whether this code is optimal in practice.

In this chapter, we firstly cast the recently proposed BER estimation problem as a two-party computation problem. From a theoretical standpoint, we proved that even when approximation and randomization are allowed the cost of this problem is $\Omega(\log n)$, where n is the length of data transmitted, which explains why both the EEC scheme and the tug-of-war sketch both need this much overhead. From a practical standpoint we presented an enhanced tug-of-war (EToW) sketch with significant additional innovations for better fitting BER estimation applications. The EToW can bring 60% or more reduction of sketch size and its performance is fully analyzable and easily tunable.

Moreover, we have systematically investigated the design space of error estimating codes, stemming from the natural question whether EEC achieves the best tradeoff between the space and estimation accuracy in estimating bit error rates. Along the path of our exploration using Fisher information analysis, we have demonstrated that EEC decoding is inefficient, and proposed a new estimator (decoder) that achieves a significantly higher accuracy. While investigating whether EEC encoding is efficient, we have developed a generalized coding framework, called generalized EEC, in which existing designs, such as EEC and EToW, are just degenerate cases. Using this unified framework, we found that

some parameterization of gEEC similar to EToW can contain around 25% more information than the pure EEC/EToW scheme.

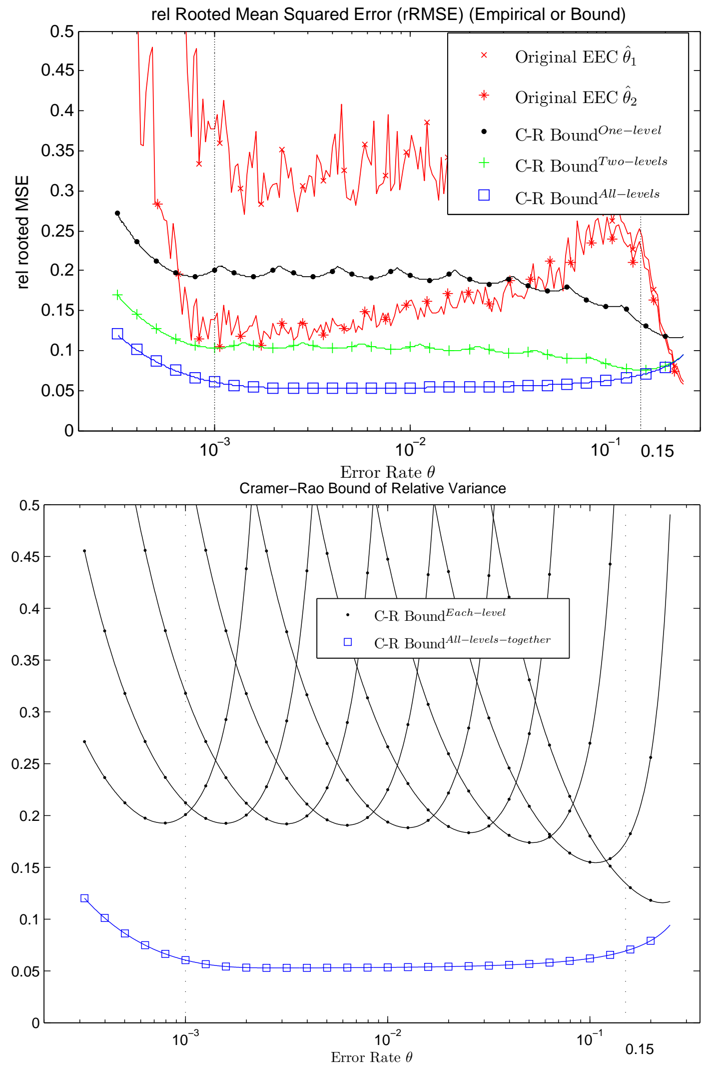


Figure 21: the empirical performance of EEC's original estimators and the associated Cramer-Rao Bound (all levels, each level, and the envelope of only one level/two levels). The EEC scheme is composed by nine levels and 32 bits each level.

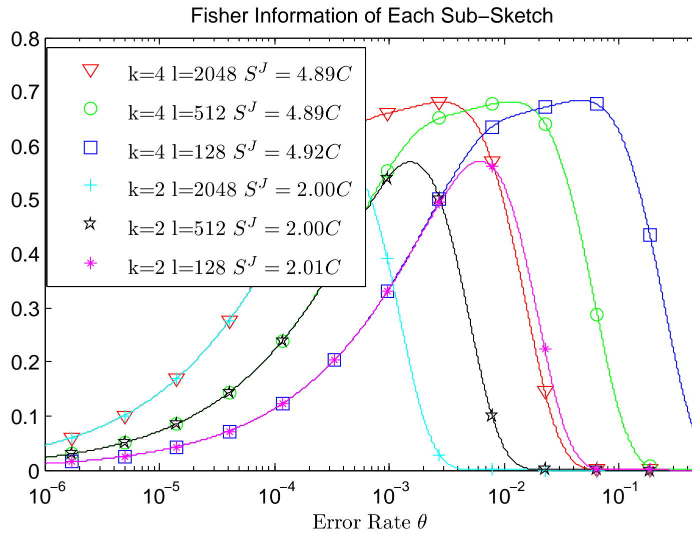


Figure 22: gEEC's Fisher information: Relationship to l and k in the case with immunity assumption

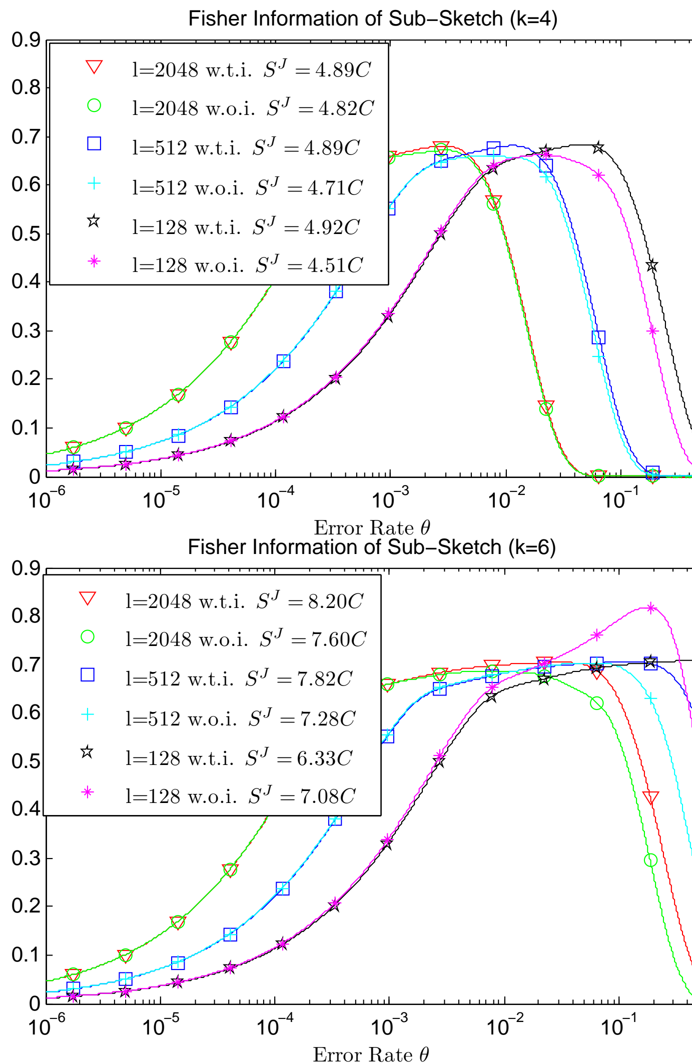


Figure 23: gEEC's Fisher information: Relationship to l and k in the cases with and without immunity (denoted as *w.t.i.* and *w.o.i.* respectively in the legend)

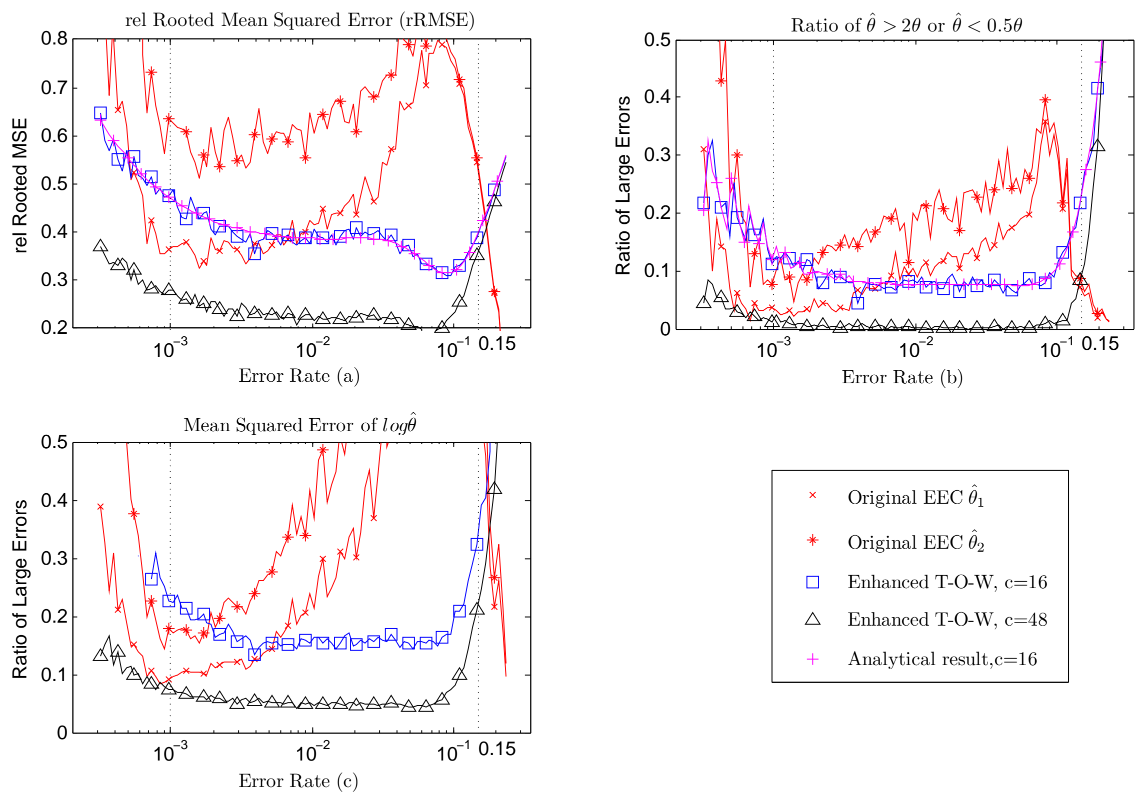


Figure 24: Experimental Results of EToW sketch

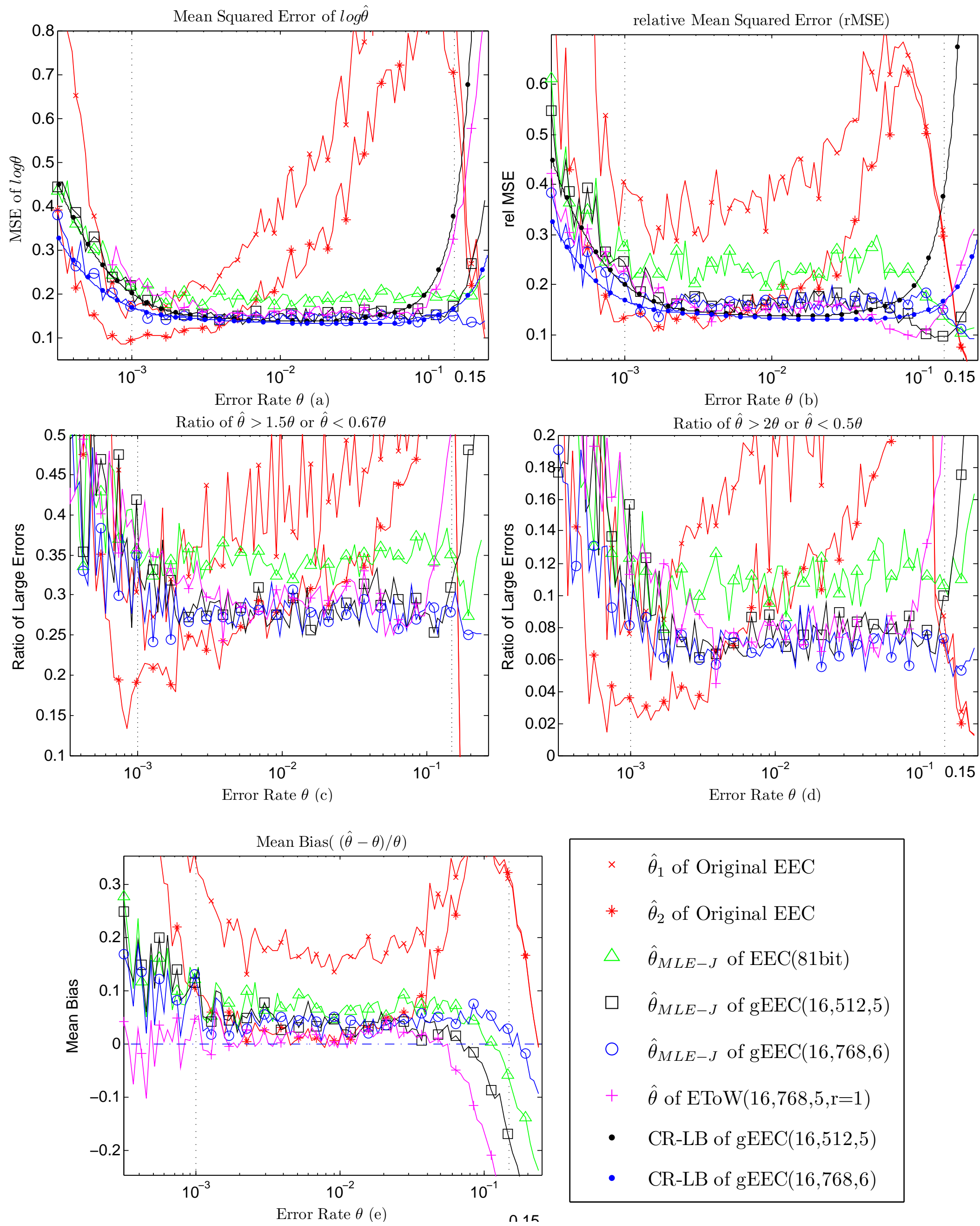


Figure 25: Empirical Results of Estimators: Compare the performance of original EEC (with original estimator), two gEEC schemes (with much smaller size, 80 and 96 bits respectively), and one EToW scheme (with 96 bits), in the case without "immunity".

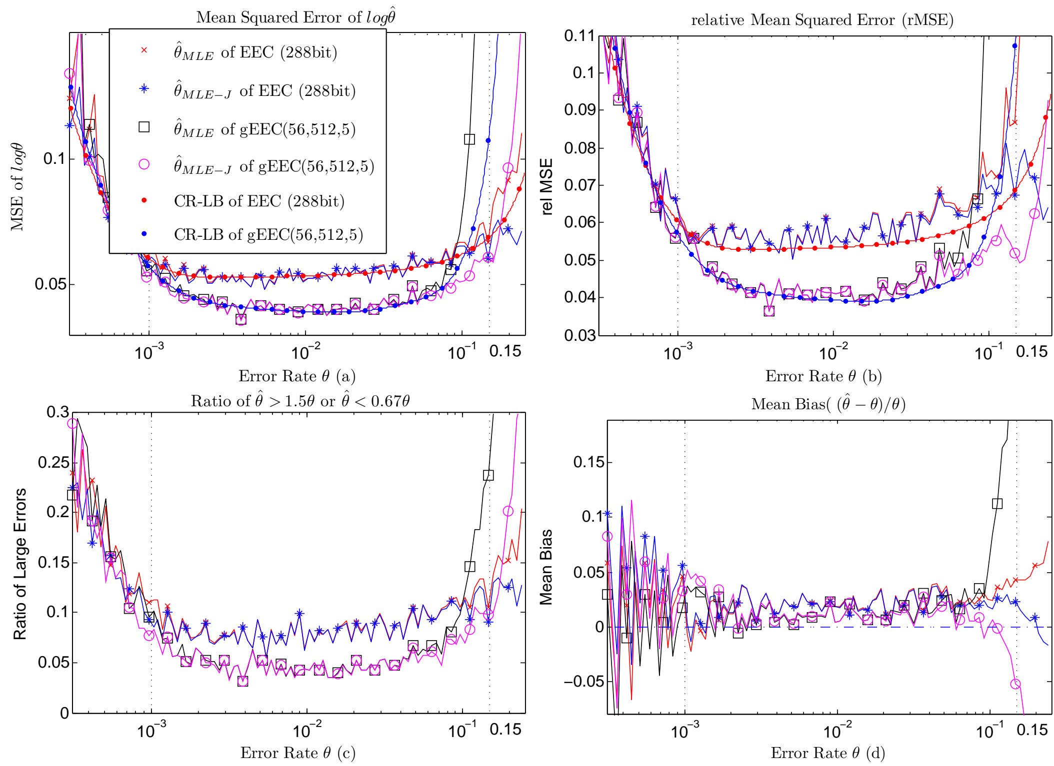


Figure 26: Empirical Results of Estimators: Compare the performance of two types of gEEC estimators (with or without Jeffrey's prior) on three schemes including the original EEC scheme, in the case without "immunity".

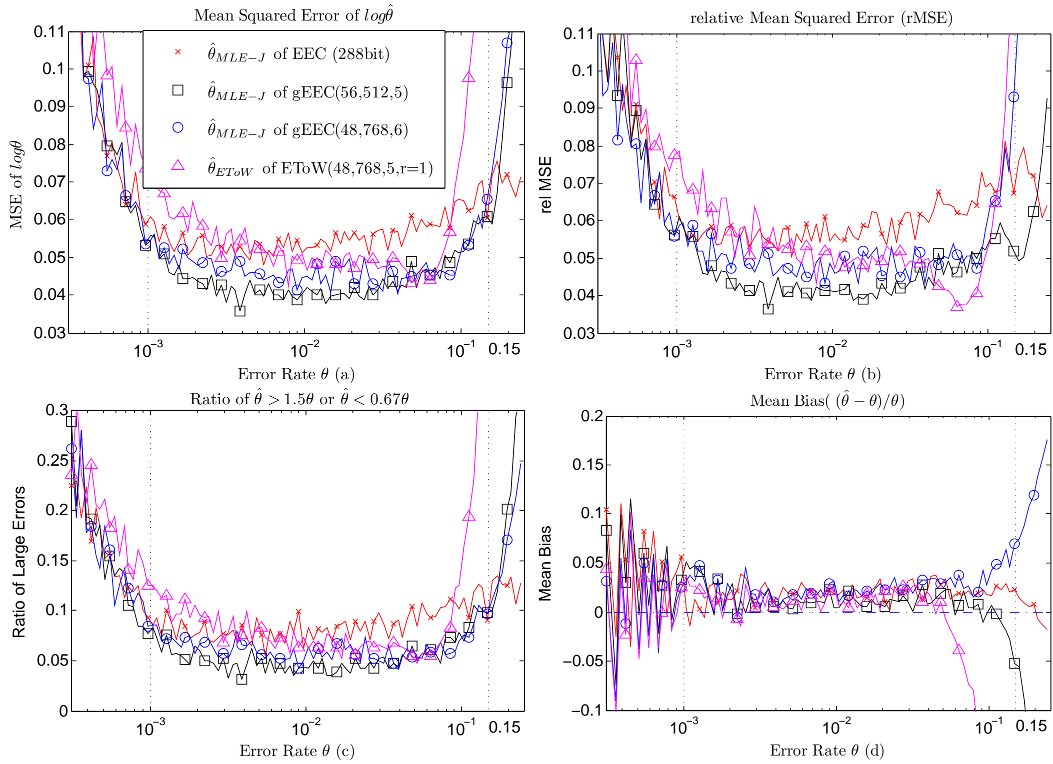


Figure 27: Empirical Results of Estimators: Compare the performance of four schemes (original EEC, gEEC(56,512,5),gEEC(48,768,6) and EToW), with almost the same size and using gEEC's estimator with Jeffrey's prior, in the case without "immunity".

CHAPTER V

CONCLUSION AND FUTURE WORKS

In this dissertation research, we have researched three network sketching problems, namely the designs and analysis of Bloom Filter, exact active counter array, and error estimating coding. All these problems require the creation and maintenance of succinct sketches of packets or flows that will later be queried by the respective applications to extract information out. The sizes of sketches is a paramount concern in all these sketching applications. and even modest reduction of sketch sizes (while delivering the same or better functionalities and accuracies) could mean considerable savings in precious on-chip memory resources for the first two applications and less transmission overhead for the last one.

We have proposed one or two techniques for each aforementioned problem, along with rigorous analyses of their performance and the closeness to the optimality. Interestingly, all these techniques exploit randomization and statistical multiplexing, in a way different than do other network applications which usually statistically multiplex traffic flows into a link with fixed bandwidth. In the rank-indexed Bloom filter and the rank-indexed counter architecture, we statistically pack variable number of fingerprints or counter segments into a fixed size bucket (Figure 9) and statistically guarantee the number of overflows from the buckets. In the improved error estimating codes, we statistically pack lower-order bits (after truncation) into a sub-sketch . Our key findings are listed as follows:

1. **Rank-Indexed Bloom Filter** Rank-Indexed Bloom Filter, is a new fingerprint hash table construction that can achieve the combined functionalities of Bloom filters, counting Bloom Filters, and several other variants. It inherits all the benefits from the fingerprint hash-table approach and hence support both deletion and query of associate value, while also has the advantage of less storage cost and better memory

locality. Since the rank-indexing technique is almost orthogonal to d-left hashing, we can combine these two techniques together to achieve more memory savings at the slight cost of decreased memory locality and increased amortized memory access cost. We have also developed rigorous analysis of our bloom filter's performance.

2. **BRICK: an exact and active counter architecture** BRICK (Bucketized Rank Indexed Counters), is a novel exact active statistics counter architecture that can very efficiently store large arrays of variable length counters entirely in SRAM while supporting extremely fast increments and lookups. The high memory (SRAM) efficiency is achieved through the rank-indexing technique and the statistical multiplexing technique similar to Rank-Indexed Bloom Filter, which by grouping a fixed number of randomly selected counters into a bucket, allows us to tightly bound the amount of memory that needs to be allocated to each bucket. The statistical guarantees of BRICK are proven using some more theory including stochastic ordering theory and probabilistic tail bound techniques. The performance is verified by simulation using real-world Internet traffic traces .

3. **Error Estimating Coding** Error estimating Coding was originally proposed in the seminal work of Chen *et al.* [13]. In this thesis, we have designed a new estimator for the original EEC solution in [13] and designed two new sketch data structures for estimating BERs with estimators, namely Enhanced Tug-of-War(EToW) sketch and the generalized EEC (gEEC) family of sketches. The performances of our proposed techniques are close to each other (with up to 25% difference) while all of them are much better than that of the original algorithm (up to 70% improvement). We have established asymptotic lower bounds for error estimating code through Fisher information and Cramer-Rao bound analyses.We found that the original EEC scheme (with original estimator) is asymptotically the optimal but not in the constant factor. Our techniques, in comparison, can match the Cramer-Rao bound asymptotically

and empirically. In addition, all schemes and estimators's performance are analyzed, proved, and verified through experiments.

As for the future work, I consider the following directions promising and interesting:

1. **Improving and extending the rank-indexing techniques** One major contribution of this thesis is the rank-indexing techniques with statistical multiplexing for Bloom filter and counter array problems. Rank-indexing is indeed a powerful indexing scheme that has much much smaller overhead than naïve pointer-based indexing. However, rank-indexing scheme does have its weakness: it incurs the overhead of memory moves during insertions and deletions. We wonder whether there can be some alternative schemes that can achieve better tradeoffs between rank-indexing and the naïve pointer-based indexing.

   We are also interested in applying the combination of rank-indexing technqiue with statistical multiplexing on other application areas where dynamic indexing and management of small chunks of memory are needed, such as the memory management for packet buffers.

2. **Comparison and combination of sampling & streaming** During the research on error estimating codes, we noticed some interesting connections between sampling and streaming. First, our results reveal an important fact that data streaming, where every bit(or item) participates in the sketch calculation, doesn't necessarily improve the results significantly. For certain applications where users might be interested only in better estimation accuracies when the parameter at issue lies within a particular range,combining sampling and streaming together might deliver even better results. Hence we are interested in extending this idea for some other data streaming problem such as the sketches in [15] and enable the users to tune the configuration of the sketch towards better space efficiency.

   Moreover, I also noticed that an interesting fact from Section 4.5.1, where I find

that the streaming of the entire packet, which can be viewed as sampling all bits without replacement, performs empirically better than sampling the same number of bits with replacement. However, sampling with replacement is much easier to be analyzed and hence employed throughout our two schemes, EToW and gEEC. We are interested in further analyzing the difference between sampling with and without replacement from an information theoretic perspective. We hope that the mathematical techniques developed in [68], where an inequality connecting sampling with and without replacement is proved, will provide a starting point for this exploration.

3. **Error Estimating Codes** In this thesis, we have proposed two error estimating coding schemes, namely EToW and gEEC, from an algorithmic perspective, and verified their performance by extensive comparison of the statistics and the CDF of theoretical and experimental results. We are interested in collaborating with some other research groups to implement and evaluate our EEC codes in some real wireless settings.

   We also noticed that, although we proved the asymptotic lower bound in Section 4.3 and had the Cramer-Rao lower bound for all proposed schemes, we have not so far proved that our gEEC sketch family either reaches or be close to the non-asymptotic lower bound. We are curious whether there are sketching techniques that can achieve higher information density (i.e. higher amount of Fisher information per sketch bit). We are also curious whether we can find an exact (i.e. non-asymptotic) lower bound for the EEC problem, even for a particular family of sketches.

REFERENCES

[1] "Intel 64 and IA-32 architectures software developer's manual, volume 2B," Nov. 2007. Available at ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf.

[2] "Software optimization guide for AMD family 10h processors," Dec. 2007. Available at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf.

[3] ALON, N., GIBBONS, P. B., MATIAS, Y., and SZEGEDY, M., "Tracking join and self-join sizes in limited storage," *J. Comput. Syst. Sci.*, vol. 64, no. 3, pp. 719–747, 2002.

[4] BLANDFORD, D. K. and BLELLOCH, G. E., "Dictionaries using variable-length keys and data, with applications," in *SODA*, pp. 1–10, 2005.

[5] BLOOM, B. H., "Space/time trade-offs in hash coding with allowable errors," *CACM*, vol. 13, no. 7, pp. 422–426, 1970.

[6] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., and VARGHESE, G., "Beyond bloom filters: from approximate membership checks to approximate state machines," in *SIGCOMM*, pp. 315–326, 2006.

[7] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., and VARGHESE, G., "An improved construction for counting bloom filters," in *ESA*, pp. 684–695, 2006.

[8] BRODER, A. Z. and MITZENMACHER, M., "Using multiple hash functions to improve ip lookups," in *INFOCOM*, pp. 1454–1463, 2001.

[9] BRODER, A. Z. and MITZENMACHER, M., "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, 2003.

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., and GRUBER, R. E., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[11] CHAZELLE, B., KILIAN, J., RUBINFELD, R., and TAL, A., "The bloomier filter: an efficient data structure for static support lookup tables," in *SODA*, pp. 30–39, 2004.

[12] CHEN, B., ZHOU, Z., ZHAO, Y., and YU, H., "Technical report: Efficient error estimating coding: feasibility and applications."

[13] CHEN, B., ZHOU, Z., ZHAO, Y., and YU, H., "Efficient error estimating coding: feasibility and applications," in *SIGCOMM*, pp. 3–14, 2010.

[14] COHEN, S. and MATIAS, Y., "Spectral bloom filters," in *SIGMOD Conference*, pp. 241–252, 2003.

[15] CORMODE, G. and MUTHUKRISHNAN, S., "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, 2004.

[16] COVER, T. M. and THOMAS, J. A., *Elements of information theory (2. ed.)*. Wiley, 2006.

[17] CVETKOVSKI, A., "An algorithm for approximate counting using limited memory resources," in *Proc of ACM SIGMETRICS*, 2007.

[18] DEGERMARK, M., BRODNIK, A., CARLSSON, S., and PINK, S., "Small forwarding tables for fast routing lookups," in *SIGCOMM*, pp. 3–14, 1997.

[19] DUBOIS-FERRIÈRE, H., ESTRIN, D., and VETTERLI, M., "Packet combining in sensor networks," in *SenSys*, pp. 102–115, 2005.

[20] ELAOUD, M. and RAMANATHAN, P., "Adaptive use of error-correcting codes for real-time communication in wireless networks," in *INFOCOM*, pp. 548–555, 1998.

[21] ESTAN, C. and VARGHESE, G., "New directions in traffic measurement and a ccounting," in *Proc. of ACM SIGCOMM*, Aug. 2002.

[22] FAN, L., CAO, P., ALMEIDA, J. M., and BRODER, A. Z., "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[23] FEIGENBAUM, J., KANNAN, S., STRAUSS, M., and VISWANATHAN, M., "An approximate l1-difference algorithm for massive data streams," *SIAM J. Comput.*, vol. 32, no. 1, pp. 131–151, 2002.

[24] FIRTH, D., "Bias reduction of maximum likelihood estimates," *Biometrika*, vol. 80, no. 1, pp. 27–38, 1993.

[25] FLAJOLET, P. and MARTIN, G. N., "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[26] GALLAGER, R. G., *Low-Density Parity-Check Codes*. MIT Press, 1963.

[27] GÖBDOI, R., "Bounds for median and 50 percentage point of binomial and negative binomial distribution," in *Metrika, Volume 41, Number 1, 43-54*, 2003.

[28] GUO, S., HE, T., MOKBEL, M. F., STANKOVIC, J. A., and ABDELZAHER, T. F., "On accurate and efficient statistical counting in sensor-based surveillance systems," *Pervasive and Mobile Computing*, vol. 6, no. 1, pp. 74–92, 2010.

[29] HOLLAND, G., VAIDYA, N., and BAHL, P., "A rate-adaptive mac protocol for multihop wireless networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, (New York, NY, USA), pp. 236–251, ACM, 2001.

[30] HUA, N., LALL, A., LI, B., and XU, J., "A simpler and better design of error estimating coding," in *IEEE INFOCOM*, 2012.

[31] HUA, N., LALL, A., LI, B., and XU, J., "Towards optimal error-estimating codes through the lens of fisher information analysis," in *ACM Sigmetrics*, 2012.

[32] HUA, N., LIN, B., XU, J. J., and ZHAO, H. C., "Brick: a novel exact active statistics counter architecture," in *ANCS*, pp. 89–98, 2008.

[33] HUA, N., ZHAO, H., LIN, B., and XU, J., "Rank-indexed hashing: A compact construction of bloom filters and variants," in *Network Protocols, 2008. ICNP 2008. IEEE International Conference on*, pp. 73 –82, oct. 2008.

[34] INDYK, P., "Stable distributions, pseudorandom generators, embeddings, and data stream computation," *J. ACM*, vol. 53, no. 3, pp. 307–323, 2006.

[35] JACOBSON, G., "Space-efficient static trees and graphs," in *30th FOCS*, pp. 549–554, 1989.

[36] JAMIESON, K. and BALAKRISHNAN, H., "Ppr: partial packet recovery for wireless networks," in *SIGCOMM*, pp. 409–420, 2007.

[37] KALYANASUNDARAM, B. and SCHNITGER, G., "The probabilistic communication complexity of set intersection," *SIAM J. Discrete Math.*, vol. 5, no. 4, pp. 545–557, 1992.

[38] KANE, D. M., NELSON, J., and WOODRUFF, D. P., "On the exact space complexity of sketching and streaming small norms," in *SODA*, pp. 1161–1178, 2010.

[39] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., and CHEN, Y., "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. of ACM SIGCOMM IMC*, Oct. 2003.

[40] KUMAR, A., SUNG, M., XU, J., and WANG, J., "Data streaming algorithms for efficient and accurate estimation of flow size distribution," in *Proc. of ACM SIGMETRICS*, 2004.

[41] KUSHILEVITZ, E. and NISAN, N., *Communication complexity*. Cambridge University Press, 1997.

[42] LEHMANN, E. and CASELLA, G., *Theory of Point Estimation, 2nd edition*. Springer, 1998.

[43] LIN, K. C.-J., KUSHMAN, N., and KATABI, D., "Ziptx: Harnessing partial packets in 802.11 networks," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, MobiCom '08, (New York, NY, USA), pp. 351–362, ACM, 2008.

[44] LIN, S. and COSTELLO, D. J., *Error Control Coding: Fundamentals and Applications, 2nd edition*. Pearson-Prentice Hall, 2004.

[45] LU, Y., MONTANARI, A., PRABHAKAR, B., DHARMAPURIKAR, S., and KABBANI, A., "Counter braids: A novel counter architecture for per-flow measurement," in *Prof. of ACM SIGMETRICS*, 2008.

[46] LU, Y., MONTANARI, A., and PRABHAKAR, B., "Detailed network measurements using sparse graph counters: The theory," in *Proceedings of 39th Allerton Conference, September 2007*.

[47] MITZENMACHER, M., "Compressed bloom filters," *IEEE/ACM Transactions on Networking*, pp. 613–620, Oct. 2002.

[48] MITZENMACHER, M. and UPFAL, E., *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[49] MORRIS, R., "Counting large numbers of events in small registers," in *Commun. ACM 21(10)*, 1978.

[50] MOTWANI, R. and RAGHAVAN, P., *Randomized Algorithms*. Cambridge University Press, 1995.

[51] MÜLLER, A. and STOYAN, D., *Comparison Methods for Stochastic Models and Risks*. Wiley, 2002.

[52] MUTHUKRISHNAN, S., "Data streams: Algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, vol. 1, no. 2, 2005.

[53] RAMABHADRAN, S. and VARGHESE, G., "Efficient implementation of a statistics counter architecture," in *Proc. ACM SIGMETRICS*, June 2003.

[54] RIBEIRO, B. F., TOWSLEY, D. F., YE, T., and BOLOT, J., "Fisher information of sampled packets: an application to flow size estimation," in *Internet Measurement Conference*, pp. 15–26, 2006.

[55] ROEDER, M. and LIN, B., "Maintaining exact statistics counters with a multilevel counter memory," in *Proc. of IEEE Globecom, Dallas, USA, 2004*.

[56] SHAH, D., IYER, S., PRABHAKAR, B., and MCKEOWN, N., "Maintaining statistics counters in router line cards," in *IEEE Micro*, 2002.

[57] SINGH, A., KONRAD, A., and JOSEPH, A. D., "Performance evaluation of udp lite for cellular video," in *NOSSDAV*, pp. 117–124, 2001.

[58] STANOJEVIC, R., "Small active counters," in *Proc of IEEE Infocom*, 2007.

[59] TUCK, N., SHERWOOD, T., CALDER, B., and VARGHESE, G., "Deterministic memory-efficient string matching algorithms for intrusion detection," in *IEEE INFOCOM*, 2004.

[60] TUNE, P. and VEITCH, D., "Towards optimal sampling for flow size estimation," in *Internet Measurement Conference*, pp. 243–256, 2008.

[61] TUNE, P. and VEITCH, D., "Sampling vs sketching: An information theoretic comparison," in *INFOCOM*, pp. 2105–2113, 2011.

[62] VÖCKING, B., "How asymmetry helps load balancing," in *FOCS*, pp. 131–141, 1999.

[63] VÖCKING, B., "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, 2003.

[64] VUTUKURU, M., BALAKRISHNAN, H., and JAMIESON, K., "Cross-layer wireless bit rate adaptation," in *SIGCOMM*, pp. 3–14, 2009.

[65] WONG, S. H. Y., LU, S., YANG, H., and BHARGHAVAN, V., "Robust rate adaptation for 802.11 wireless networks," in *MOBICOM*, pp. 146–157, 2006.

[66] WOODRUFF, D. P. personal communication, 2011.

[67] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., and LUND, C., "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and application," in *Proc. of ACM SIGCOMM IMC*, Oct. 2004.

[68] ZHAO, H. C., WANG, H., LIN, B., and XU, J. J., "Design and performance analysis of a dram-based statistics counter array architecture," in *ANCS*, pp. 84–93, 2009.

[69] ZHAO, Q., KUMAR, A., WANG, J., and XU, J., "Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices," in *Proc. of ACM SIGMETRICS*, June 2005.

[70] ZHAO, Q., XU, J., and LIU, Z., "Design of a novel statistics counter architecture with optimal space and time efficiency," in *Proc. of ACM SIGMETRICS*, June 2006.

VITA

Nan Hua received his Bachelor of Engineering and Master of Engineering from Department of Electronic Engineering, Tsinghua University, Beijing, China in 2005 and 2007 respectively. Nan joined College of Computing, Georgia Institute of Technology, as a Ph.D. student in August 2007. His thesis work was conducted under the able guidance of Dr. Jun (Jim) Xu.