2018

ULTRA-FAST AND MEMORY-EFFICIENT LOOKUPS FOR CLOUD, NETWORKED SYSTEMS, AND MASSIVE DATA MANAGEMENT

Ye Yu
University of Kentucky, ye.yu@uky.edu
Author ORCID Identifier:
https://orcid.org/0000-0003-4541-6918
Digital Object Identifier: https://doi.org/10.13023/ETD.2018.222

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Ye Yu, Student

Dr. Chen Qian, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

ULTRA-FAST AND MEMORY-EFFICIENT LOOKUPS FOR CLOUD, NETWORKED SYSTEMS, AND MASSIVE DATA MANAGEMENT

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By

Ye Yu

Lexington, Kentucky

Directors: Dr. Chen Qian , Professor of Computer Science
and Dr. Jinze Liu , Professor of Computer Science

Lexington, Kentucky

ABSTRACT OF DISSERTATION

ULTRA-FAST AND MEMORY-EFFICIENT LOOKUPS FOR CLOUD, NETWORKED SYSTEMS, AND MASSIVE DATA MANAGEMENT

Systems that process big data (e.g., high-traffic networks and large-scale storage) prefer data structures and algorithms with small memory and fast processing speed. Efficient and fast algorithms play an essential role in system design, despite the improvement of hardware. This dissertation is organized around a novel algorithm called *Othello Hashing*. Othello Hashing supports ultra-fast and memory-efficient key-value lookup, and it fits the requirements of the core algorithms of many large-scale systems and big data applications. Using Othello hashing, combined with domain expertise in cloud, computer networks, big data, and bioinformatics, I developed the following applications that resolve several major challenges in the area.

Concise: Forwarding Information Base. A Forwarding Information Base is a data structure used by the data plane of a forwarding device to determine the proper forwarding actions for packets. The polymorphic property of Othello Hashing the separation of its query and control functionalities, which is a perfect match to the programmable networks such as Software Defined Networks. Using Othello Hashing, we built a fast and scalable FIB named *Concise*. Extensive evaluation results on three different platforms show that Concise outperforms other FIB designs.

SDLB: Cloud Load Balancer. In a cloud network, the layer-4 load balancer servers is a device that acts as a reverse proxy and distributes network or application traffic across a number of servers. We built a software load balancer with Othello Hashing techniques named *SDLB*. SDLB is able to accomplish two functionalities of the SDLB using one Othello query: to find the designated server for packets of ongoing sessions and to distribute new or session-free packets.

MetaOthello: Taxonomic Classification of Metagenomic Sequences. Metagenomic read classification is a critical step in the identification and quantification of microbial species sampled by high-throughput sequencing. Due to the growing popularity of metagenomic data in both basic science and clinical applications, as well as the increasing volume of data being generated, efficient and accurate algorithms are in high demand. We built a system to support efficient classification of taxonomic sequences using its k-mer signatures.

SeqOthello: RNA-seq Sequence Search Engine. Advances in the study of functional genomics produced a vast supply of RNA-seq datasets. However, how to quickly query and extract information from sequencing resources remains a challenging problem and has been the bottleneck for the broader dissemination of sequencing efforts. The challenge resides in both the sheer volume of the data and its nature of unstructured representation. Using the Othello Hashing techniques, we built the SeqOthello sequence search engine. SeqOthello is a reference-free, alignment-free, and parameter-free sequence search system that supports arbitrary sequence query against large collections of RNA-seq experiments, which enables large-scale integrative studies using sequence-level data.

KEYWORDS: Othello Hashing, Forwarding Information Base, Software Load Balancer, Taxonomy Classification, Sequence Search

YE YU
Student's Signature

MAY 29, 2018
Date

ULTRA-FAST AND MEMORY-EFFICIENT LOOKUPS FOR CLOUD, NETWORKED SYSTEMS, AND MASSIVE DATA MANAGEMENT

By

Ye Yu

CHEN QIAN
Co-Director of Dissertation

JINZE LIU
Co-Director of Dissertation

MIROSLAW TRUSZCZYNSKI
Director of Graduate Studies

MAY 29, 2018
Date

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Dr. Chen Qian for his patient guidance. Five years ago, when I decided to join his group, I could never imagine the facinating experience over this journey. Dr. Qian has always been resourceful and supportive. His advice has been priceless for me. I also thank him for his understanding and continuous support after he moved to the University of California Santa Cruz.

Dr. Jinze Liu has guided and supported my research in the past years. The applications of Othello in large-scale bioinformatics data processing started coincidentally, and turned out to be fantastic. I was amazingly fortunate to have both Dr. Chen Qian and Dr. Jinze Liu's support and guidance. I could not have reached this point without them.

I would also like to thank my Ph.D. Committee Members, Dr. Ken Calvert, Dr. Zongming Fei, Dr. Sen-ching Samson Cheung for their invaluable advice and directions throughout the years.

I am grateful to Dr. Ken Calvert, Dr. Jurek Jaromczyk, and Dr. Ying Zhang, for providing valuable suggestions and support in my job search.

Thank goes to my collaborators from academic institutions and the industry, for the valuable discussion we had during the past years. Especially, I would like to acknowledge Dr. Ying Zhang, Dr. Djamal Belazzougui, Dr. Qin Zhang, and Dr. Jishen Zhao for their suggesion and input in my research.

I would like to thank Dr. Jurek Jaromczyk and Dr. Miroslaw Truszczynski for their support and guidance in my years at the University of Kentucky.

Thanks also go to the members of the Qian's lab and Liu's lab. I cherish the opportunity to work with my fellows and to establish a friendship with them: Dr. Xinan Liu, Jinpeng Liu, Eamonn Manager, Xin Li, Huazhe Wang, Shouqian Shi, and Yi Zhang.

My Ph.D. would not have been possible without my family and friends' support and encouragement. I am very grateful to my parents for their love and for providing me a relaxed and supportive environment. I would also like to thank the local Chinese Christian community for the support and company.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

Systems that process big data (e.g., high-traffic networks and large-scale storage) are facing emerging challenges in generating, transmitting, storing, and processing large-scale data. Hence, they prefer data structures and algorithms with small memory and fast processing speed. Despite the improvement of hardware, the efficiency of algorithms always plays an essential role in system design.

Specifically, key-value lookup is a fundamental mechanism that presents in almost all computational systems or applications. A key-value lookup data structure stores the mapping between a set of keys to the corresponding values. During each query, a key is specified and the query result is the corresponding value of this key. Intuitively, a hash table may serve as a key-value lookup table in many applications. However, in some data-intensive or lookup intensive applications, the lookup speed of hash table still becomes the performance bottleneck. Hence, for real-world systems, the need for an efficient data structure for key-value query lookup never cease.

This dissertation is organized around an algorithm called *Othello Hashing*. Othello Hashing supports ultra-fast and memory-efficient key-value lookup, and it fits the requirements of the core algorithms of these large-scale systems and big data applications. Othello achieves significant speed improvement in lookups of network addresses, data IDs, or genome sequences, using significantly smaller memory compared to recent solutions. By bringing efficient algorithms and data structures for key-value lookup for practical problems in building real-world systems, the works presented in this dissertation tackle challenging issues of the distributed networking systems, big data, and Bioinformatics.

1.1 Background

1.1.1 Background in cloud and networked systems

Significant efforts have been devoted to the investigation and deployment of new network technologies in order to simplify network management and to accommodate emerging network applications, especially cloud and big data applications. In the past several decades, the computer networking industry has shown enormous interests in the development and application of Software Defined Networks (SDN) [2, 3]. SDN technologies make the network more programmable; meanwhile, it brings possibilities for network function virtualization (NFV)[4]. A wide range of programmable network initiatives [5], network management infrastructure [6, 7] and tools [3] have emerged during the evolution of computer network systems. Meanwhile, although the performance and capacity of hardware have been boosting rapidly in the past several decades, there is still an eternal question about how to effectively perform tasks (e.g., network packet forwarding) with limited hardware resources.

One example of such issue is the *Forwarding Information Base (FIB) explosion* problem, an inevitable issue caused by the prevailing applications of flat-names in networked systems. Though different proposals of new network technologies focus on a wide range of issues, one consensus of most new network designs is the separation of network identifiers and locators [8], which are combined in IP addresses in the current Internet. Instead of IP, flat-name or namespace-neutral architectures have been proposed to provide persistent network identifiers. A flat or location-independent namespace has no inherent structure and hence imposes no restrictions to referenced elements [9].

Forwarding Information Base (FIB) is a data structure, typically a table, that is used to determine the proper forwarding actions for packets, at the data plane of a forwarding device (e.g, switch or router). Unlike IP addresses, location-independent names are difficult to aggregate in the FIB, due to the lack of hierarchy and semantics. The increasing

population of network hosts results in huge FIBs and their continuing fast growth.

On the other hand, the increasing line speed requires the capability of fast forwarding. To support multiple 10Gb Ethernet links, a FIB may need to perform hundreds of millions of lookups per second. Existing high-end switch fabrics use fast memory, such as TCAM or SRAM, to support intensive FIB query requests. However, as discussed in many studies [10–12], fast memory is expensive, power-hungry, and hence very limited on forwarding devices. Therefore, achieving *fast queries* with *memory-efficient* FIBs is crucial for the new network architectures that rely on location-independent names.

Another issue is how to effectively perform *Load Balance in Mobile Edge Computing*. Although mobile hardware continues to improve, it is still relatively resource-constrained compared to static computing hardware. To provide the resource of computation, storage, and bandwidth to massive mobile computing devices, using powerful back-end servers in a remote cloud is a common solution. However, Many modern applications such as augmented reality (AR) and real-time monitoring/control are latency-sensitive and may suffer the long round-trip delay to the cloud. Hence, Mobile Edge Computing (MEC) has been proposed to shift computing and storage capacities from the remote cloud to the network edge [13, 14].

In MEC, The nodes that provide the resource to mobile devices are called *MEC hosts*. A MEC network deals with the data traffic from mobile devices to MEC hosts, serving various applications and purposes. The *MEC load balancer* (MEC-LB) is a network function deployed between the mobile devices and MEC hosts, which assigns a MEC host as the destination for every packet from a mobile device.

The network traffic (packets) in MEC can be classified into two categories: the stateful packets, which should be forwarded to some designated hosts according to the packet identifies; the stateless packets, which should be randomly assigned and forwarded to one of the candidate MEC hosts according to a probabilistic distribution. The MEC-LB should be able to handle both stateful packets and stateless packets simultaneously. However, existing

solutions of cloud load balancers [15–17] cannot be directly applied to MEC-LB because they can only handle either stateful or stateless packets. Meanwhile, the limited computational resource on MEC devices and the programmability requirements also calls for a memory-efficient, software-based, and portable solution. Hence, software load balancing in MEC remains a challenging but essential issue in MEC.

Although the application scenarios are different in the two above examples of the FIB explosion issue and the MEC load balancing issue, the demands of an efficient key-value lookup method are similar. The detailed approaches of applying the Othello Hashing in the real-world systems are subject to the specific use case, which is discussed later in Chapter 3 and Chapter 4 in this work.

1.1.2 Background in Bioinformatics

Deoxyribonucleic acid (DNA) molecules store the hereditary information that instructs the development and functioning of organisms. Such genetic information flows in biological systems following the Central dogma of molecular biology [18].

In the recent years, next-generation sequencing technologies (NGS) has emerged and became a prevailing technology that advances the study of molecular biology. These technical advances further improve the studies of functional genomics over the past decade. NGS plays an increasingly prominent role in biological studies, for example, RNA-seq technologies reveals the presence and quantity of RNA, which has become the dominant technology in transcriptome profiling. The unique capability in deep sequencing RNA transcripts enables the researchers to uncover important gene expression changes on a transcriptome level with unprecedented details.

While the rapid development of sequencing technology has exponentially scaled the size of genomics studies, as shown in Figure 1.1, the reduction in sequencing costs has exceeded Moore's law. Many computational methods have been developed to assist the analysis of the data. While the progress in computational technologies such as CPU, memory,

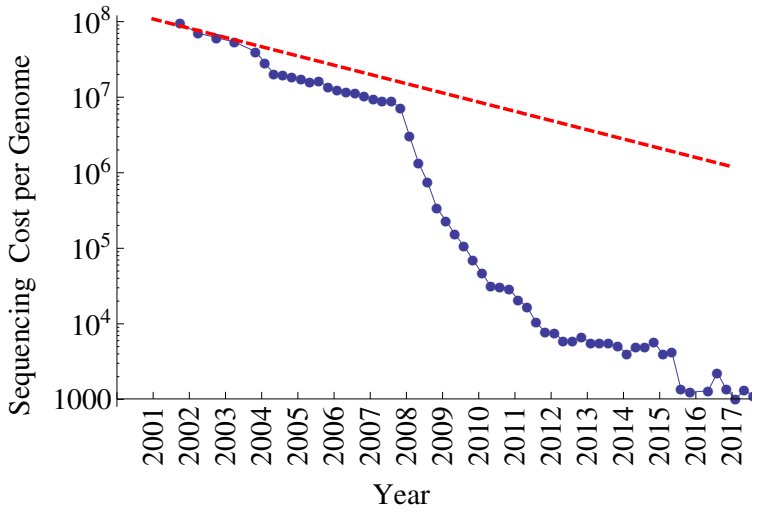


Figure 1.1: The reduction in sequencing cost is outpacing Moore's Law. Red line: hypothetical data reflecting Moore's Law. Data source: National Human Genome Research Institute (NHGRI) [1]

storage, has been struggling to follow the Moore's Law, the need of efficient approaches to process, store, and query the RNA-seq data has become more critical to many bioinformatics research projects.

To date, there is a vast supply of genomics datasets, including datasets that are public or shared within the community (SRA[19], TCGA[20], ICGC[21], NCBI[22], etc.), as well as private datasets generated in individual research labs. Thanks to the development of distributed system and cloud computing infrastructure, researchers are now able to conduct the analytic workflow of these datasets using cloud infrastructure [23–25]. In order to analyze large-scale genomic data, researchers have been making efforts to utilize the superb features of cloud computing, such as elastic and parallel computing, to parallel execute the sub procedures in the analytic workflow [26, 27]. However, few efforts have been made to the development of algorithm or system foundations of these procedures.

We observe that there is a great need for effective index lookup algorithm in the area of large-scale Bioinformatics data storage and query. In a data storage system where each record contains the data of a corresponding key, an index of the key returns the location of the corresponding data. An index lookup is equivalent to a key-value query. Index lookup

is also equivalent to classification query. For a collection of a disjointing set of keys, given a key that belongs to one of the sets, a classification query of this key identifies the set that contains this key. We identify two use cases where such algorithm is critical to the applications.

One example use case lies in the area of *Taxonomic Classification of Metagenomic Sequences*. Metagenomics is the study of genomic content obtained in bulk from an environment of interest, such as the human body [28], seawater [29], or acidic mine drainage [30]. One of the major computational challenges in the analysis of metagenomic data is the classification of each sequencing read into the most-specific biological taxon to which sequence conservation supports its assignment. The atomic operation for such classification operation of sequencing read is a classification query of each small segment of the sequencing reads over the collection of taxons.

Another use case is the *Sequence Search Query on Large Collection of Experiments*. Advances in the study of functional genomics produced a vast supply of RNA-seq datasets in the past decade [19, 20]. However, quickly querying and extracting information from the sequencing resources remains a challenging problem and the bottleneck for the broader dissemination of sequencing efforts. Most sequencing databases only support metadata searches[19, 21, 22, 31], which fails to extract base-by-base k-mer (k-mer is the atomic unit for sequence data) coverage profiles of a query sequence across large sequencing datasets. Such fine-grained information is essential to identify mutations, distinguish transcript isoforms and predict a sequences expression level. In order to achieve detailed global characterization of genomic features at the k-mer level, the database must be able to report the present/absence information of individual k-mer. While such information is stored as compressed data entries in a large data store, an efficient indexing algorithm is crucial in the process of retrieving such information.

While the key-value lookup algorithm serves as the central component of both the above applications, we also have taken various practical issues into the consideration for the actual

design of the Bioinformatics applications. These details are presented in Chapter 5 and Chapter 6.

1.2 Dissertation Statement

This dissertation aims to investigate the algorithmic properties of Othello Hashing, to evaluate its performance, and to explore its applications in large-scale enterprise networks, cloud systems, and Bioinformatics.

1.3 Dissertation Contribution

This dissertation is organized around a novel algorithm called *Othello Hashing*. The main functionality of Othello Hashing is to provide a classification key-value query. For a set of keys $S = \{s_1, s_2, \cdots, s_n\}$ (names, identifies, etc.), and a corresponding list of fixed-length non-negative integer values $T = (t_1, t_2, \cdots, t_n)$, an Othello data structure $\boldsymbol{O}(S,T)$ maintains the mapping τ from the keys to the values.

The Othello data structure supports the query operation. Given a key s_i that is present in the set S, $s_i \in S$, the query result on Othello $\tau(s)$ returns the corresponding value t_i, i.e., $\tau(s_i) = t_i$. For any key s' that is not specified in the set S, namely *alien* key, $s' \notin S$, the query result $\tau(s')$ returns a deterministic value that follows a particular probability distribution. As a comparison, for a hash table maintains a key-value mapping, a query of an alien key yields a result reporting such key does not exist in the table.

In many of the applications for networked systems and big data, the computational resource of the construction and update operation is usually abundant, while the query procedure usually faces limited computational resources and strict real-time performance requirements. For example, in a programmable network, the resource-abundant control plane is responsible for handling network dynamics, while the data plane switches forward the packets. The limited resource and strict performance requirement for the data plane demands a query mechanism that is optimized for memory efficiency and query speed.

Othello Hashing is tailored to fit these applications, the query

The journey of exploring the applications of Othello Hashing starts from the high-traffic computer networking for cloud application, which later expanded to the big data and bioinformatics area. Using Othello hashing, combined with domain knowledge in cloud, computer networks, big data, and bioinformatics, I developed the following systems and applications that resolve several major challenges in the area.

Concise: Forwarding Information Base [32]. A Forwarding Information Base is a data structure, typically a table, used to determine the proper forwarding actions for packets at the data plane of a forwarding device (e.g., switch or router). The increasing size of FIBs, due to the ever-growing number of connected network devices or entities, causes many problems including large memory cost (hence more expensive switches/routers), slow table lookups, and coarse-grained flow management. Meanwhile, the increasing line speed also requires a fast and small FIBs. The polymorphic property enables the separation of Othello's query and control functionalities, which is a perfect match to the programmable networks such as Software Defined Networks. I built a fast and scalable FIB named Concise. Extensive evaluation results on three different platforms show that Concise outperforms other FIB designs. Compared to existing FIB designs for name switching, Concise supports much faster name lookup using significantly smaller memory (2x $\sim$ 4x faster with only 10% $\sim$ 30% memory size compared to state-of-art solutions).

SDLB: Cloud Load Balancer [33]. In a cloud, the load balancer is a device that acts as a reverse proxy and distributes network or application traffic across a number of servers. Load balancers are widely used in cloud computing and mobile edge computing to increase the capacity and reliability of applications. A layer-4 load balancer serves the following two functions. 1) A lookup to find the designated server if a packet belongs to an ongoing session. 2) If the packet is new or session-free, the load balancer selects one of the available servers according to the capacity of the servers. I built a software load balancer with Othello Hashing techniques named SDLB, which enables the load balancer

to perform the two functions simultaneously using one query on the table. Evaluation results show that the system is faster by 4x to 10x and uses much less ($< 50\%$) memory, than the current widely-used load balancer designs.

MetaOthello: Taxonomic Classification of Metagenomic Sequences [34]. Metagenomic read classification is a critical step in the identification and quantification of microbial species sampled by high-throughput sequencing. Although many algorithms have been developed to date, they suffer significant memory and computational costs. Due to the growing popularity of metagenomic data in both basic science and clinical applications, as well as the increasing volume of data being generated, efficient and accurate algorithms are in high demand. We built a system to support an efficient classification of taxonomic sequences using its k-mer signatures. MetaOthello is an order-of-magnitude faster than the current state-of-the art algorithms Kraken and Clark and requires only one-third of the RAM.

SeqOthello: RNA-seq Sequence Search Engine [35]. Advances in the study of functional genomics produced a vast supply of RNA-seq datasets. However, how to quickly query and extract information from sequencing resources remains a challenging problem and has been the bottleneck for the broader dissemination of sequencing efforts. The challenge resides in both the sheer volume of the data and its nature of unstructured representation. We carefully designed the algorithms so that it enables sequence search on a vastly compressed data domain. Using the Othello Hashing techniques, we built the first sequence search index constructed on the scale of TCGA data. SeqOthello requires only five minutes to conduct a global survey of 11,658 fusion events against 10,113 TCGA Pan-Cancer RNA-seq datasets on a standard computer with 19.1 GB memory space. The query recovers 92.7% of tier-1 fusions curated by TCGA Fusion Gene Database and further reveals 270 novel fusion occurrences, all of which present as tumor-specific. The entire index is only 76 GB, achieving a 700:1 compression ratio relative to the original sequencing data and making it extremely portable. By providing a reference-free, alignment-free,

and parameter-free sequence search system, SeqOthello will enable large-scale integrative studies using sequence-level data, an undertaking not previously practicable for many individual labs.

The rest of this dissertation is organized as follows. Chapter 2 presents the Othello Hashing algorithm. The following Chapters presents the four different applications of Othello Hashing: Chapter 3 presents the Concise Forwarding Information Base, Chapter 4 presents the SDLB software load balancer, Chapter 5 presents the MetaOthello taxonomy classification system, Chapter 6 presents the SeqOthello sequence search engine. Finally, 7 summarizes the work and discuss the potential future works of Othello Hashing.

Chapter 2. Othello Hashing Algorithm

In this chapter, we describe the Othello Hashing algorithm.

We first describe the background and related work of minimal perfect hashing in Section 2.1. We give the definitions of Othello Hashing and the notations in Section 2.2. The operations that Othello supports are discussed in Section 2.3. The implementation considerations are presented in Section 2.4. The properties for alien query on Othello is discussed in 2.5. We provide an extensive evaluation of Othello in 2.6. Finally, we summarize the properties in 2.7.

2.1 Background and Related Work

The data structure used in this work, Othello Hashing, is built upon the studies on minimal perfect hashing. [36–39, 39–43]

A perfect hash function h of a set S maps the set to hash values without conflict. i.e., for any $s_1, s_2 \in S$, $s_1 \neq s_2$, $h(s_1) \neq h(s_2)$. A minimal hash function maps a set of $n = |S|$ keys into n consecutive integral numbers $\{0, 1, \cdots, n-1\}$. The minimal perfect function of set S generates a complete order of the elements in S, hence minimal perfect hashing techniques are usually used for building effective index of large data sets.

In particular, MWHC [37] is able to generate order-preserving minimal perfect hash functions using a random hypergraph. MWHC is also presented as Bloomier Filter in [44, 45]. This approach is independently rediscovered in [42, 43]. A r-uniform hypergraph H is a pair $H = (V, E)$ where V is the set of nodes and E is the set of hyper-edges. Each hyper-edge $e \in E$ is a set of r elements in V. These works choose different r values to make

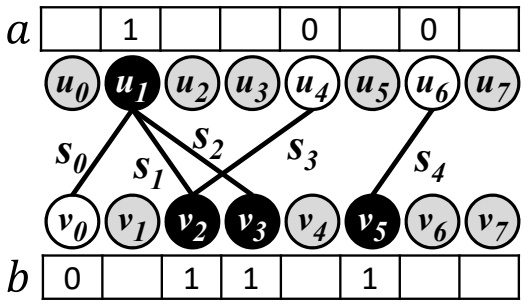
Figure 2.1: Example of the board game Othello. Image source [47], use under CC-BY-SA 3.0.

trade off between the memory overhead and the computational overhead. For example, in order to achieve minimal memory size $r = 3$ was used in [43].

The differences between Othello and these studies include: (1) Othello uses a bipartite graph instead of a general random hypergraph. This design allows much simpler concurrency control mechanism. (2) The original researches on MWHC and Bloomier Filter are designed for static scenarios. The dynamic update mechanism was not presented, which could be more complicated in practice. (3) Othello is optimized for real applications. It is designed under the polymorphic data structure model and performs different functionalities by the control structure and query structure. Othello aims to support fast flat key-value classification query, while MWHC is for finding minimum perfect hash functions [37] and Bloomier Filter is designed for approximate evaluation queries [44].

The Name *Othello* We name the data structure by *Othello* because the underlying algorithm shares some interesting properties with the board game *Reversi*. The game is marketed under the trademark *Othello*. Researchers on combinatorial game theory has been interested in the Othello Game and found intriguing properties of the game[46].

The board game Othello is a strategy board game in which two players take turns placing tokens on the board. A photo of an Othello board is shown in Figure 2.1. The tokens in



s	$h_a(s)$	$h_b(s)$	$\tau(s)$
s_0	1	0	1
s_1	1	2	0
s_2	1	3	0
s_3	4	2	1
s_4	6	5	1

Figure 2.2: Example of 1-Othello of $n = 5$ keys with $m_a = m_b = 8$. Left: Bipartite graph G and bitmaps A and B. Right: the hash values and $\tau(s)$ values for five keys $s_0, s_1, s_2, s_3, s_4, s_5$.

the game are small disks which are white on one side and black on the other side. During the game, the disks are placed on the board, which may be later turned over by the players. This is much similar to the arrays A and B in the Othello Hashing data structure when $l = 1$, in which the values 0 and 1 are placed in the arrays which can be later 'flipped' by update operations.

2.2 Definitions: l-Othello

For a set of keys (names, identifies, etc.) $S = \{s_1, s_2, \cdots, s_n\}$, and a corresponding list of l-bit non negative integer values $T = (t_1, t_2, \cdots, t_n)$, where $0 \leq t_i < 2^l$, we use the notation $\boldsymbol{O}(S,T)$ to describe the following data structure: Othello $\boldsymbol{O}(S,T)$ maintains a mapping $\tau : S \to \{0, 1, \cdots, 2^l - 1\}$ that satisfies for any $i \in \{1, 2, \cdots, n\}, \tau(s_i) = t_i$.

An l-Othello $\boldsymbol{O}(S,T)$ is a seven-tuple $\langle m_a, m_b, h_a, h_b, A, B, G \rangle$, defined as follows.

- Integers m_a and m_b, describing the size of Othello.
- A pair of uniform random hash functions $\langle h_a, h_b \rangle$, mapping keys to integer values $\{0, 1, \cdots, m_a - 1\}$ and $\{0, 1, \cdots, m_b - 1\}$, respectively.
- Arrays A and B of l-bit integers. The lengths are m_a and m_b respectively.
- A bipartite graph G. During Othello construction and update, G is used to determine the values in A and B.

Figure 2.2 shows an l-Othello example where $l = 1$. We require that $m_a = \Theta(n)$, $m_b =$

$\Theta(n)$, and $m_a m_b > n^2$. We provide two options to determine the values m_a and m_b. 1) m_a is the smallest power of 2 such that $m_a \geq 1.33n$ and $m_b = m_a$. 2) m_a is the smallest power of 2 such that $m_a \geq 1.33n$ and m_b is the smallest power of 2 such that $m_b \geq n$. A user may choose either option. The difference is that for Option 1 we establish a rigorous proof of constant update time and for Option 2 we establish the proof with a constraint on n. However Option 2 provides slightly better empirical results.

l-Othello supports a query operation as follows. For a key s, it computes $\tau(s) \in \{0,1,\cdots,2^l-1\}$. If $s \in S$ (i.e., $s = s_i$ for some i), $\tau(s_i) = t_i$. If $s \notin S$, $\tau(s)$ returns a value in $\{0,1,\cdots,2^l-1\}$ that is determined by the content of A and B. The values of A and B are determined during Othello construction, so that $\tau(s)$ can be computed by:

$$\tau(s) = A[h_a(s)] \oplus B[h_b(s)]$$

Here, $\oplus$ is the *exclusive or* (`XOR`) operation.

As shown in Figure 2.2, when $l = 1$, the values $\tau(s)$, $A[h_a(s_i)]$, and $B[h_b(s_i)]$ are all 1-bit values. The arrays in A and B of 1-Othello satisfy:

- If $t_i = 0$, $A[h_a(s_i)] = B[h_b(s_i)]$;
- If $t_i = 1$, $A[h_a(s_i)] \neq B[h_b(s_i)]$.

2.3 Othello Operations

Othello is maintained via the following operations.

- `construct`(S,T): Construct an l-Othello. (Section 2.3.1)
- `add`(s,t): add a new key s into the set S and specify the corresponding query result be t. (Section 2.3.2)
- `alter`(s,t): For a key $s \in S$, change the corresponding value in Othello, so that the query result $\tau(s)$ returns t after this operation. (Section 2.3.3)

- `delete`(s): For a key $s \in S$, remove s from set S and no longer maintain the corresponding t value. (Section 2.3.4)

2.3.1 Construction

The `construct` operation for l-Othello takes a list of keys S and the list of corresponding values T as input. The output is an l-Othello $\boldsymbol{O}(S,T) = \langle m_a, m_b, h_a, h_b, A, B, G \rangle$.

Here, G is a bipartite graph used to determine the hash function pair and the values of A and B. $G = (U,V,E)$. $|U| = m_a$, $|V| = m_b$. A vertex $u_i \in U$ or $v_j \in V$ corresponds to bit $A[i]$ or $B[j]$. Each edge in E represents a key. There is an edge $(u_i, v_j) \in E$ if and only if there is a key $s \in S$ such that $h_a(s) = i$ and $h_b(s) = j$.

We use the case $l = 1$ for example. For each vertex that is associated with at least one edge, the corresponding bit is set to 0 or 1. A vertex associated with bit 0 is colored in white and a vertex associated with bit 1 is colored in black. For vertices that have no associated edges, the value of the corresponding bits can be set to 0 or 1 arbitrarily, because they do not affect any $\tau(s)$ value for $s \in S$. In order to assign correct values of A and B, Othello requires G to be *acyclic*.

The construction algorithm consists of two phases.

- *Phase I: Selecting the hash function pair.*

  In this phase, Othello finds a hash function pair $\langle h_a, h_b \rangle$. We assume there are many candidate hash functions and will discuss the implementation in Sec. 2.4.2. In each round, two hash functions are chosen randomly and G is accordingly generated. We use Depth-First-Search (DFS) on G to test whether it includes a cycle, which takes $O(n)$ time. The order in which the edges are visited during the DFS, i.e, the DFS order of the edges, is recorded to prepare for the second phase. Note that if two or more keys generate edges with the same two endpoints, we will consider as if there is a cycle. If G is cyclic, the algorithm will select another pair of hash functions until an acyclic G is found.

- *Phase II: Computing the bitmaps.*

  In this phase, we assign values for the two bitmaps A and B. First, the values in A and B are marked as undefined. Then, we execute the followings for each $e = (u_i, v_j)$ in the DFS order of the edges: Let s be the key that generates e. If none of $A[i]$ and $B[j]$ has been assigned, let $A[i] \leftarrow 0$ and $B[j] \leftarrow \tau(s)$. If there is only one of $A[i]$ and $B[j]$ has been assigned, we can always assign an appropriate value to the other one, such that $A[i] \oplus B[j] = \tau(s)$. As G is acyclic, following the DFS order, we will never see an edge such that both $A[i]$ and $B[j]$ have values.

We show the pseudocode of l-Othello construction in Algorithm 1.

Input: Keys $S = \{s_1, s_2, \cdots, s_n\}$, values $T = (t_1, t_2, \cdots, t_n)$
Output: An Othello structure $\boldsymbol{O}(S,T) = \langle m, h_a, h_b, A, B, G \rangle$
begin

```
1   select m value according to n = |S|.
                /* Phase I: decide hash function pair */
2   repeat
3   |  Randomly select hash function h_a, h_b.
    until GeneratedGraphIsAcyclic(S, h_a, h_b).
                    /* Phase II: Compute bitmaps */
4   Compute G = (U, V, E) using h_a, h_b and S.
5   Execute Depth-First-Search on G.
6   (e_1, e_2, ..., e_n) ← the DFS order of E.
7   Mark all A[i], B[j] (0 ≤ i, j < m) as unassigned.
8   for w = 1, 2, ..., n do
9   |  s ← the corresponding key for e_w
10  |  v ← the corresponding t value for s
11  |  i ← h_a(s); j ← h_b(s).
12  |  if both A[i] and B[j] are unassigned then
13  |  |  A[i] ← 0; B[j] ← v.
14  |  else if A[i] is unassigned then
15  |  |  A[i] ← B[j] ⊕ v.
16  |  else                                  /* B[j] isunassigned */
17  |  |  B[j] ← A[i] ⊕ v.
    |  end
    end
end
```

Algorithm 1: Othello `construct` procedure

Note that the edges of G are only determined by S and the hash function pair $\langle h_a, h_b \rangle$. If we find G to be cyclic for a given S and a pair $\langle h_a, h_b \rangle$, we shall use another pair $\langle h_a, h_b \rangle$ to make G acyclic. We show that for a randomly selected pair of hash functions $\langle h_a, h_b \rangle$, the probability of G to be acyclic is very high:

Theorem 1. *Given set of keys S, $n = |S|$. Suppose h_a, h_b are randomly selected from a family of fully random hash functions. $h_a : S \to \{0, 1, \cdots, m_a - 1\}$, $h_b : S \to \{0, 1, \cdots, m_b - 1\}$. Then the generated bipartite graph G is acyclic with probability $\sqrt{1 - c^2}$ when $n \to \infty$, where $c = \frac{n}{\sqrt{m_a m_b}}$, $c < 1$.*

When G is acyclic, we say that $\langle h_a, h_b \rangle$ is a *valid hash function pair* for S. We prove Theorem 1 using the technique described in [48].

Proof. Let $G = (U, V, E)$ be a bipartite random graph with $|U| = m_a$, $|V| = m_b$, $|E| = n$, where each edge is independently taken at random with probability $\frac{n}{m_a m_b}$. Let $\mathscr{C}_{2\ell}$ be the set of cycles of length 2ℓ ($\ell \geq 1$) in the complete bipartite graph K_{m_a, m_b}. A cycle in $\mathscr{C}_{2\ell}$ is a sequence of 2ℓ distinct vertices chosen from U and V. Hence,

$$|\mathscr{C}_{2\ell}| = \frac{1}{2\ell}(m_a)_\ell (m_b)_\ell,$$

where $(m)_\ell = m(m-1)\cdots(m-\ell+1)$. Meanwhile, As each edge in G is selected independently, each cycle in $\mathscr{C}_{2\ell}$ occurs in G with probability $(\frac{n}{m_a m_b})^{2\ell}$.

As proved in [48], the number of cycles of length 2ℓ in G converges to a Poisson distribution with parameter $\lambda_{2\ell}$. For $n \to \infty$,

$$\begin{aligned} \lambda_{2\ell} &= p^{2\ell} |\mathscr{C}_{2\ell}| \\ &= (\frac{n}{m_a m_b})^{2\ell} \frac{1}{2\ell} (m_a)_\ell (m_b)_\ell \to \frac{1}{2\ell} \frac{n^{2\ell}}{(m_a m_b)^\ell} \end{aligned}$$

Let $c = \frac{n}{\sqrt{m_a m_b}}$ we have $\lambda_{2\ell} \to \frac{1}{2\ell} c^{2\ell}$ as $n \to \infty$.

The number of cycles of any even length in G, represented as a random variable $\mathscr{X}$,

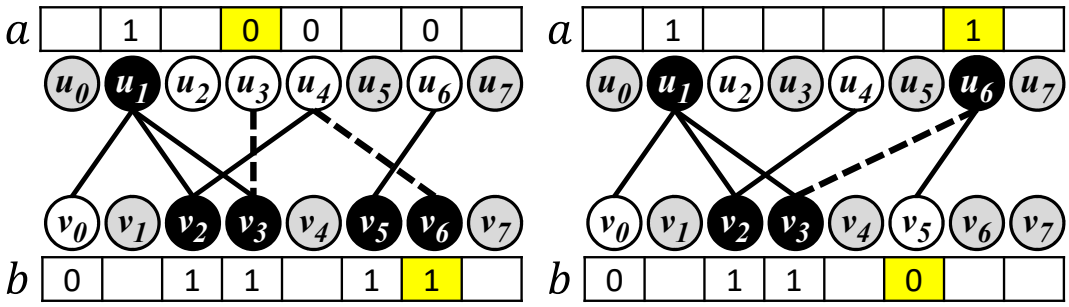


Figure 2.3: Example of adding keys into Othello. Dashed edges: added keys. Highlighted cells: modified values in a and b. Left: e adds isolated nodes to existing connected components, Right: e joins two existing non-trivial connected components.

converges to a Poisson distribution with parameter λ_e, where

$$\lambda_e = \sum_{\ell=1}^{\infty} \lambda_{2\ell} = -\frac{1}{2}\ln(1-c^2).$$

Therefore, the probability that G contains no cycle is

$$\Pr(\mathscr{X} = 0) = e^{-\lambda_e} = \sqrt{1-c^2}.$$

□

When $c \leq 0.75$ (i.e, $n \leq 0.75m$), $\sqrt{1-c^2} \geq 0.66$. Hence the expected number of rounds to find an acyclic G in Phase I is $\frac{1}{\sqrt{1-c^2}} \leq 1.51$ when $c < 0.75$. The time complexity is $O(n)$ in each round. The second phase takes $O(n)$ time to visit n edges and assign values of A and B. Hence, the total expected time of `construct` is $O(n)$.

2.3.2 Key addition

To `add` a key s with value t using `add`(s,t), the graph G and two arrays A and B should be changed in order to maintain the correct result $\tau(s)$.

The algorithm first computes the edge $e = (u,v)$ to be added to G for s, $u = u_{h_a(x)}$, $v = v_{h_b(x)}$. Note that G can be decomposed into connected components. e must fall in one of the following cases:

- *Case I:* u and v belong to the same connected component `cc`. Adding e to G will introduce a cycle. In this case, we have to re-select a hash function pair $\langle h_a, h_b \rangle$ until a valid hash function pair is found for the new key set $S \cup \{k\}$. The `construct` algorithm is used to perform this process.

- *Case II:* u and v are in two different connected components. As shown in Figure 2.3, the edge e either (1) adds an isolated node to a connected component in G; (2) joins two connected components in G. Combining the two connected components and the new edge, we have a single connected component that is still acyclic. As discussed in Sec. 2.3.1, it is simple to find a valid coloring plan for an acyclic connected component. Hence, the values of A and B can also be set properly. In fact, at least one of the two connected components can keep the existing value assignments.

Complexity Analysis

We now compute the time complexity of `add` using three theorems. In particular, we will show that the time complexity of the `add` operation is $O(1)$. The important parameter that governs the complexity of an insertion is the *susceptibility* of the graph G, which is defined as the expected size of the connected component that contains a randomly chosen node, and is denoted by $\chi(G)$.

We give a closed-form estimation for $\chi(G) = \frac{1}{1-p}$ where $p = \frac{n(m_a+m_b)}{2m_a m_b}$, and prove that $\chi(G)$ has a constant upperbound $E[\chi(G)] \leq 4$. As stated before, there are two options in choosing values m_a and m_b. In Option 1, $m_a = m_b$ and in Option 2, $m_a = m_b$ or $m_a = 2m_b$. We are able to compute the closed-form formulae for $\chi(G)$ when $m_a = m_b$. For the case $m_a = 2m_b$, we give a looser upper bound. The numerical estimation shows that the upper bound $E[\chi(G)] \leq 4$ is true for both of the two situations where $m_a = m_b$ and $m_a = 2m_b$.

For the sake of analysis we let $\mathscr{G}_A(m_a, m_b, n)$ be a random acyclic graph generated using the same process as $\mathscr{G}(m_a, m_b, n)$ except that an edge is not added if it introduces a cycle in the graph. It could also be generated by repeatedly generating graphs $\mathscr{G}(m_a, m_b, n)$ until we

get an acyclic graph. It is evident that this random graph model corresponds to the graphs constructed and maintained by Othello.

For the case $m_a = m_b$ we show Theorem 2. For the case $m_a = 2m_b$ we show Theorem 3. Theorem 4 concludes that the time complexity of `add` is $O(1)$.

Theorem 2. *Suppose we have a random graph $\mathcal{G}_A(m_a, m_b, n)$ where $m_a = m_b$ and we randomly select a node w in $\mathcal{G}_A$. Let $\texttt{cc}(w)$ be the connected component containing w. Then the expected value of $|\texttt{cc}(w)|$ is $\frac{m_a}{m_a - n}$ as $n \to \infty$.*

Proof. Let $\chi(G) = E[|\texttt{cc}(w)|]$ where w is randomly selected from G and $|\texttt{cc}(w)|$ denotes the number of nodes in $\texttt{cc}(w)$. In [49, Lemma 1], it was proved that for a random sparse graph $\mathcal{G}(m_a, m_a, n)$ with n edges, we have $\chi(G) = \frac{2m_a}{2m_a - 2n}$ when $n \to \infty$ given that $n < 0.999m_a$. We will show that the same bound holds for a graph $\mathcal{G}_A(m_a, m_a, n)$. It is well known that the largest connected component in a random graph with n edges and m nodes with $n \leq 0.99 \cdot m/2$ has size $O(\log n)$ with probability $1 - \frac{1}{n^{10}}$ [49].

We now generate a graph $\mathcal{G}_A(m_a, m_a, n)$ by generating the edges one by one. If an edge (v, w) makes the graph cyclic, then we do not add it, but instead put it into a set S. Let E be the set of n edges in the generated acyclic graph G_1. Then graph G_2 with the set of edges $E \cup S$ will clearly be a graph $\mathcal{G}(m_a, m_a, n')$ with $n' = n + O(\log^2 n) \leq 0.999m/2$. Now we have that $\chi(G_1) \leq \chi(G_2)$ and $\chi(G_2) = \frac{2m_a}{2m_a - 2n'} \to \frac{2m_a}{2m_a - 2n}$ when $n \to \infty$. □

Theorem 3. *For a random graph $\mathcal{G}_A(m_a, m_b, n)$ where $m_a = 2m_b$, $n \leq 0.65m_b$, and randomly select a node w in $\mathcal{G}$. Let $\texttt{cc}(w)$ be the connected component containing w. Then the expected value of $|\texttt{cc}(w)|$ is $O(1)$.*

Proof. Again let $\chi(G) = E[|\texttt{cc}(w)|]$ where w is randomly selected from G. We generate a graph $\mathcal{G}_A(m_a, m_b, n)$ with $n \leq 0.65m_b$ as follows. Let V_a with $|V_a| = m_a$ be the set of nodes on the left side, and V_b with $|V_b| = m_b$ be the set of nodes on the right side. We generate edges one by one from random graph $\mathcal{G}_A(m_a + m_b, n)$, and reject an edge (v, w) if either $(v \in V_a \wedge w \in V_a)$ or $(v \in V_b \wedge w \in V_b)$. The probability of accepting an edge is thus $\frac{4}{9}$.

We stop the generation when we have finished generating the n edges, and we denote the resulting graph by G_1. We let G_2 be the graph obtained by adding all the rejected edges back to G_1. It is clear that $\chi(G_1) \leq \chi(G_2)$. Moreover, G_2 is a random graph $\mathscr{G}_A(m_a+m_b,n')$ with $n' = \frac{9}{4}n \pm O(\sqrt{n})$ with probability $1-\frac{1}{n^{10}}$. According to Theorem 3.3(i) in [50], for a graph $G_3 = \mathscr{G}(m_a+m_b,n')$:

$$\begin{aligned}\chi(G_3) &\leq \frac{m_a+m_b}{m_a+m_b-2n'} = \frac{3m_b}{3m_b-2\cdot\frac{9}{4}n} \\ &\leq \frac{3\frac{n}{0.65}}{3\frac{n}{0.65}-2\cdot\frac{9}{4}n} = O(1).\end{aligned}$$

We can use the same argument as in the proof of Theorem 2 to show that the susceptibility for a graph $\mathscr{G}_A(m_a+m_b,n')$ is the same as for a graph $\mathscr{G}(m_a+m_b,n')$ which concludes the proof. □

Theorem 4. *Assuming h_a, h_b are randomly selected from a family of fully random hash functions, an insertion into an Othello with n existing keys will take constant amortized expected time when $m_a = m_b$, or when $m_a = 2m_b$ and $n \leq 0.65m_b$.*

Proof. In the algorithm described in Section. 2.3.2, during an insertion, we have to add an edge that connects a randomly selected node $u \in U$ to another randomly selected node $v \in V$. We will first bound the amortized expected cost of insertions that fall in *Case I* and then the induced cost of insertions that fall in *Case II*. Let $|\mathsf{cc}(w)|$ be the size of the connected component that contains node w. Let $|\mathsf{cc}_b(w)|$ be the number of nodes in $\mathsf{cc} \cup V$. $|\mathsf{cc}_b(w)| < |\mathsf{cc}(w)|$.

The probability that node v falls in the same connected component as node w is $\frac{|\mathsf{cc}_b(w)|}{m_b} \leq \frac{|\mathsf{cc}(w)|}{m_b}$ which is the probability of reconstruction. Since the reconstruction takes expected $O(n)$ time, the amortized expected time cost for reconstruction is $\frac{|\mathsf{cc}_b(w)|}{m_a} \cdot O(n) = O(|\mathsf{cc}(w)|) = O(1)$.

For *Case II*, the cost is clearly $O(|\mathsf{cc}(w)| + |\mathsf{cc}(v)|) = O(1)$, since we have to traverse the connected component that results from merging the two connected components that

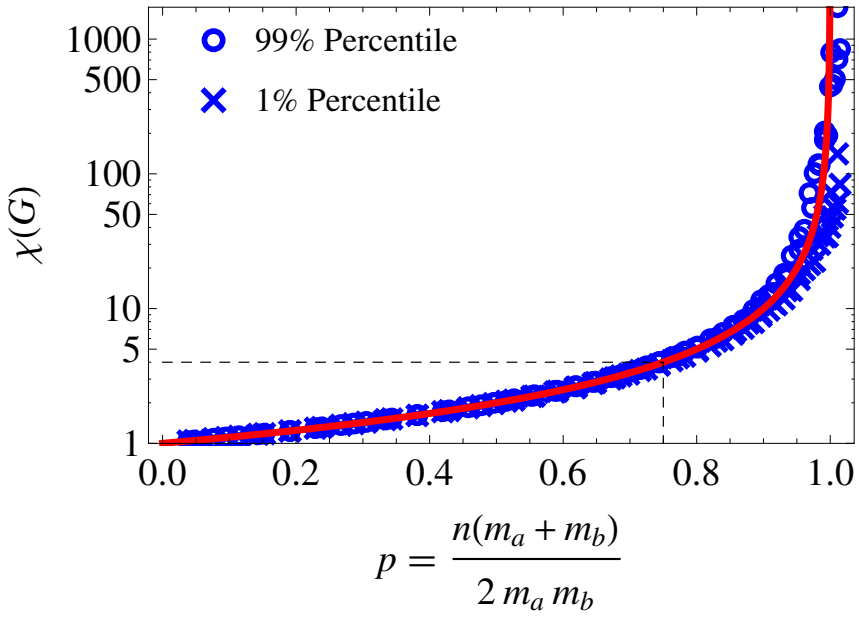


Figure 2.4: $\chi(G)$ of acyclic graphs vs parameter p. Red curve: $\frac{1}{1-p}$

contain w and v. □

Note we have a rigorous proof for Option 1 but Option 2 provides slightly better empirical results. It is reasonable to conjecture that Theorem 4 also holds for $m_a = 2m_b$ without the constraint $n \leq 0.65m_b$.

Numerical estimation of $\chi(G)$:

We conjecture that $\frac{1}{1-p}$, where $p = \frac{n(m_a+m_b)}{2m_am_b}$ is a good estimation for $\chi(G) = E[|\mathtt{cc}(w)|]$, and present numerical simulation to support our conjecture. We generate acyclic bipartite graphs with random m_a, m_b, and n values (within the range 10K $\sim$ 1M). Then we compute their $\chi(G)$ value. For a particular $p = \frac{n(m_a+m_b)}{2m_am_b}$ value, we randomly sample at least 500 graphs with different m_a,m_b, and n. In Figure 2.4, we plot the 1-th and 99-th percentile of $\chi(G)$.

As shown in Figure 2.4, when p is not so close to 1, the sampled $\chi(G)$ values are very close to $\frac{1}{1-p}$. When p grows larger, the sampled $\chi(G)$ values tend to grow slower than $\frac{1}{1-p}$. Hence we conclude that $\frac{1}{1-p}$ is a good upper bound for $\chi(G)$. In Othello, $\frac{4}{3}n \leq m_a < \frac{8}{3}n$,

$n \leq m_b < 2n$. m_a and m_b must be powers of 2. For this choice of parameters we can see that $p = \frac{n(m_a+m_b)}{2m_a m_b} \leq 0.75$ and so $\chi(G) \leq 4$ which is a small constant.

This estimated value $\chi(G) = \frac{1}{1-p}$ is in coherence with the evaluation results on Othello updates shown in Section 2.6.3.

Remarks on susceptibility of random bipartite graphs Consider a bipartite graph $G = (U,V,E)$ with k connected components $\mathtt{cc}_1, \mathtt{cc}_2, \cdots, \mathtt{cc}_k$. When we randomly select a node in G, the probability that the chosen node is in $\mathtt{cc}_i$ is $\frac{|\mathtt{cc}_i|}{|U|+|V|}$. Hence, the expected size of the component containing a randomly selected node w is

$$\mathrm{E}[|\mathtt{cc}|] = \sum_{i=1}^{k} \frac{|\mathtt{cc}_i|}{|U|+|V|} \cdot |\mathtt{cc}_i| = \frac{1}{|U|+|V|} \sum_{i=1}^{k} |\mathtt{cc}_i|^2$$

This value is also called the *susceptibility* of graph G, denoted as $\chi(G)$.

We use the results in [51] to estimate the $\chi(G) = \mathrm{E}[|\mathtt{cc}|]$ value for bipartite graph $G = (U,V,E)$. The following description is a directly application of the results presented in [52]. The meaning of the notation is listed as follows.

- Inhomogeneous random graph model $G^{\mathscr{V}}(n,\kappa)$: it generates a random graph with expected n vertices, where the vertices are generated in $\mathscr{V}$, the edges are generated following probability values defined by kernel κ. This is first presented in [51], page 4.

- Type space $(\mathscr{S},\mu)$: S is a set of *types* of vertices in a random graph, and μ is the probability measure. For bipartite graph S has two elements, $S = \{1,2\}$. In the inhomogeneous random graph model, a vertex is of type s with probability $\mu(s)/\mu(S)$. This is first presented in [51], page 4.

- Vertex space $\mathscr{V} = (\mathscr{S},\mu,(\mathrm{x})_n)$: A formal model that describes the vertices in the random bipartite graph. Here, $(\mathrm{x})_n$ is a list of n elements in S. This concept is presented in [52], page 7.

- Kernel κ is a function $\kappa : \mathscr{S} \times \mathscr{S} \to [0,1]$. $\kappa(s_1, s_2)$ describes the probability of there being an edge connecting two vertices of type s_1 and s_2. This term is presented in [52], page 7.

Consider a random graph generated with the general inhomogeneous model $G^{\mathscr{V}}(m_a + m_b, \kappa)$ as described in [52], where $\mathscr{V} = (\mathscr{S}, \mu, (\mathrm{x})_{m_a+m_b})$. The measure space $(\mathscr{S}, \mu)$ is defined as

$$\mathscr{S} = \{1,2\}; \mu(1) = \frac{m_a}{m_a + m_b}, \mu(2) = \frac{m_b}{m_a + m_b}.$$

$\kappa : \mathscr{S} \times \mathscr{S} \to [0, \infty)$ is defined as

$$\kappa(1,1) = \kappa(2,2) = 0$$

$$\kappa(1,2) = \kappa(2,1) = \frac{n(m_a + m_b)}{2m_a m_b} = p.$$

.

It is easy to verify $G^{\mathscr{V}}(m_a + m_b, \kappa)$ generates a random bipartite graph $G' = (U', V', E')$, where $\mathrm{E}[|U'|] = m_a$, $\mathrm{E}[|V'|] = m_a$ and $\mathrm{E}[|E'|] = n$.

T_κ is the integral operator in $L^2(\mathscr{S}, \mu)$,

$$T_\kappa = \begin{pmatrix} 0 & p \\ p & 0 \end{pmatrix}$$

According to Theorem 3.3(i) in [51],

$$\chi(G) = \langle (I - T_\kappa)^{-1} \cdot 1, 1 \rangle_\mu = \frac{1}{1-p}$$

Othello size growth.

After adding a key into Othello, $n = |S|$ grows and may violate $m_a \geq 1.33n$ and $m_b \geq n$. $\chi(G)$ also grows. Hence, Othello may choose to reconstruct the graph G in order

to guarantee very low amortized time overhead of future operations. However, Othello works correctly as long as G is acyclic, even when $m_a < 1.33n$ or $m_b < n$. Hence, Othello does not deal with the requirement on m_a and m_b explicitly for additions. Although the $\chi(G)$ value may grow as more keys are added to Othello, it is always smaller than 10 in our experiments. The expected time to add a key to Othello is still $O(1)$ in practice.

When adding a new key falling in Case I, the values of m_a and m_b will be updated by `construct`, which guarantees $m_a \geq 1.33n$ and $m_b \geq n$.

2.3.3 Change the corresponding value of a key

Operation `alter`(s,t) is used to change the content of the arrays so that $\tau(s)$ returns t after the operation. Note that this only applies for $s \in S$.

Let $\delta = A[h_a(s)] \oplus B[h_b(s)] \oplus t$, its easy to verify that $A[h_a(s)] \oplus (B[h_b(s)] \oplus \delta) = t$, because

$$A[h_a(s)] \oplus (B[h_b(s)] \oplus \delta) = A[h_a(s)] \oplus B[h_b(s)] \oplus A[h_a(s)] \oplus B[h_b(s)] \oplus t = t$$

Hence, in order to adjust the $\tau(s)$ value, a possible approach is to modify $B[h_b(s)]$ so that $B[h_b(s)] \leftarrow (B[h_b(s)] \oplus \delta)$. After the change $\tau(s) = t$.

The rest question is to adjust the values in A and B so that $\tau(s)$ for other $s \in S$ does not change. Note that $s \in S$, and $e = (u_{h_a(s)}, v_{h_b(s)}) \in E$. The edge e belongs to some connected component $e \in \mathtt{cc}$. $\mathtt{cc}$ can also be viewed as two sub connected components $\mathtt{cc}_x$ and $\mathtt{cc}_y$ joined by edge e. Without loss of generality assume $v_{h_b(s)} \in \mathtt{cc}_y$. Then, an approach to adjust the A and B values is to modify the corresponding positions in A and B associated to nodes in $\mathtt{cc}_y$. i.e.,

$$A[i] \leftarrow A[i] \oplus \delta \text{ for all } u_i \in \mathtt{cc}_y$$

$$B[j] \leftarrow B[j] \oplus \delta \text{ for all } v_j \in \mathtt{cc}_y$$

It is easy to verify that all edges $(u_i, v_j) \in \mathtt{cc}_y$ has the elements on its both ends modified, so that $A[i] \oplus B[j]$ remains the same after the operation.

For $l = 1$, this approach is equivalent to "flip" the colors of all vertices at one side of e, i.e., to change 0 to 1, and to change 1 to 0. The amortized time cost is $O(1)$.

2.3.4 Key deletion

`delete`(s) can be done by simply removing the edge $(u_{h_a(s)}, v_{h_b(s)})$ from G. The bitmaps A and B are not modified because the values of $\tau(s)$ after deleting s do not matter anymore. The time complexity is $O(1)$.

2.4 Implementation Considerations

2.4.1 Query structure and control structure

Each Othello is a seven-tuple $\langle m_a, m_b, h_a, h_b, A, B, G\rangle$. Note that for a query on Othello, only the first six elements are necessary for computing the τ value. The information stored in G is not needed for the query operation. Hence, we let the switches only maintain the six-tuple $\langle m_a, m_b, h_a, h_b, A, B\rangle$ in their local memory, namely the *Query structure*. Storing this six-tuple takes $(m_a + m_b)l + O(1)$ bits of memory space. The time cost for each query of Othello is equal to the sum of the cost of computing two hash values, two memory accesses for the two bitmaps, and one `XOR` arithmetic operation.

In comparison, in order to have full access and control of the Othello Hashing data structure, one would choose to maintain seven-tuple, namely the *Control Structure*. For example, in a managed network system, The controller is responsible for maintaining the FIB of the switches in the network, and hence it maintains the control structure. The switches execute the queries on the query structures.

2.4.2 Selection of Hash functions

The hash function pair is critical for system efficiency. Ideally, h_a and h_b should be chosen from a family of fully random and uniform hash functions. Similar to the implementation of CuckooSwitch [53], we apply a function $H(k, \texttt{seed})$ to generate the hashes in our implementation. Here, H is a particular hashing method and $\texttt{seed}$ is a 32-bit integer. We let $h_a(k) = H(k, \texttt{seed}_a)$ and $h_b(k) = H(k, \texttt{seed}_b)$. Thus, $\langle h_a, h_b \rangle$ is uniquely determined by a pair of integers $\langle \texttt{seed}_a, \texttt{seed}_b \rangle$.

The proper hashing method $H()$ is platform-dependent. Othello Hashing on Intel x86_-64 platforms uses the CRC32c function Using a few arithmetic instructions, the CRC32c value can be effectively mapped to $\{0, 1, \cdots, 2^t - 1\}$ for integer $t \leq 32$. For robust and faster hash results, which is then effectively mapped to a t-bit integer value where $m_a = 2^t$ or $m_b = 2^t$. Evaluation shows that CRC32c demonstrates desirable performance in practice.

2.5 Othello Properties for Alien Key Queries

2.5.1 Preliminaries

An alien key is defined as a key that is not included during the construction of an l-Othello. Here we show the properties of alien queries on l-Othello. We also show how we may leverage the randomness of alien assignment to predict an alien key within the l-Othello itself.

We first discuss the query result for an alien key s on Othello $\boldsymbol{O}(S, T)$ for $l = 1$, and then we extend it to $l > 1$. When $l = 1$, the query result $\tau(s')$ is an 1-bit value. Each element in A or B is a 1-bit value. For a query of an alien key $s' \notin S$, l-Othello still returns a value $\tau(s') \in \{0, 1\}$. For alien keys, $\tau(s') = A[h_a(s')] \oplus B[h_b(s')]$. Let a_0 and a_1 be the fraction of 0s and 1s in the bitmaps A respectively, *i.e.,*

$$a_0 = \frac{|\{t|A[t] = 0\}|}{m_a}$$

$$a_1 = \frac{|\{t|A[t]=1\}|}{m_a}$$

. Similarly, b_0 and b_1 are the fractions in B. Suppose h_a and h_b are uniformly distributed random hash functions, and s' is an arbitrary key in the universal set, then $\tau(s')$ returns 1 with probability $p_1 = a_0b_1 + a_1b_0$. Similarly, $\tau(s')$ returns 0 with probability $p_0 = a_0b_0 + a_1b_1$.

For l-Othello $\boldsymbol{O}(S,T)$, a similar property also holds. Let ${}^{\boldsymbol{O}}p_t$ be the probability that the query of an alien key returns exactly t. (We use the left superscript to distinguish that this probability is computed for Othello $\boldsymbol{O}$.) $\tau(s') = t$ indicates $A[h_a(s')] \oplus B[h_b(s')] = t$. Note that h_a and h_b are uniform random hash functions and are not correlated. Hence,

$${}^{\boldsymbol{O}}p_t = \Pr[\tau(s') = t] = \sum_{x=0}^{2^l-1} a_x b_{x\oplus t}$$

Here a_x is the fraction of elements has value x in a and $b_{x\oplus t}$ is the fraction of elements has value $x \oplus t$ in B. i.e.,

$$a_x = \frac{|\{t|A[t]=x\}|}{m_a}$$

$$b_x = \frac{|\{t|B[t]=x\}|}{m_b}$$

Given a particular l-Othello, we can always compute p_t values for all $t = 0, 1, \cdots, 2^l - 1$ using time $O(2^{2l} + n)$. These p_t values are affected by the occurrence frequency of each l-bit integer, namely a_x and b_x for all $0 \le x < 2^l$. In most cases, the value ${}^{\boldsymbol{O}}p_0$ are relatively larger than other ${}^{\boldsymbol{O}}p_t$ ($t \neq 0$). (Section 2.5.2).

In some cases, these values are not uniformly distributed, which may result in imbalance among ${}^{\boldsymbol{O}}p_t$. Under such circumstances, we can always tune these values by flipping the bitmaps of a connected component in the bipartite graph without changing $\tau(s)$ for $s \in S$.

2.5.2 Detecting alien queries with probability

We shows that for a Othello constructed using the algorithm described in Section 2.3.1, the query result $\tau(s')$ for an alien key s' returns 0 with a relatively larger probability. Note that the query result on an l-Othello is an l-bit integer in the range $\{0,1,2,\cdots,2^l-1\}$, One possible approach to enable Othello to detect alien queries is to mark the range of valid query results. Say, for a classifier that classifies keys into w categories, we use the value $1,2,\cdots,w$ as the *valid* query results. Some of the alien queries would be detected when $\tau(s')$ returns 0 or some other value $\tau(s') > w$.

Theorem 5. *For an Othello $\boldsymbol{O}(S,T)$ constructed with $n = |S|$ keys. ${}^{\boldsymbol{O}}p_0 > 0.223$ as $n \to \infty$.*

Proof. We give an estimated lower bound on ${}^{\boldsymbol{O}}p_0$, which is the probability that $\tau(s')$ returns 0. Array A of the Othello contains m_a elements. Each key $s \in S$ is mapped to an index of array A computed by $h_a(s)$, where h_a is a uniform random hash function. Assuming the number of keys, n, is large, the possibility of an index in A not being hit by any of the $h_a(s)$ values is

$$\lim_{n\to\infty} a_0 = \left(1-\frac{1}{m_a}\right)^n = e^{-\frac{n}{m_a}}$$

An analogous statement holds for array B. Note that $m_a = 2^{\lceil n \rceil}$ and $m_b = 2^{\lceil \frac{4}{3}n \rceil}$. We have

$$1 < n\left(\frac{1}{m_a}+\frac{1}{m_b}\right) \le 1.5$$

$${}^{\boldsymbol{O}}p_0 = \sum_{x=0}^{2^l-1} a_x b_x > a_0 b_0 \to e^{-\frac{n}{m_a}} e^{-\frac{n}{m_b}} = e^{-n(\frac{1}{m_a}+\frac{1}{m_b})} > e^{-1.5} = 0.223$$

□

2.5.3 Othello as a deterministic randomizer

An important property of Othello is that when $s \notin S$, $\tau(s)$ returns an arbitrary value. The property was considered a weakness of a key-value table. However, it is a perfect feature that can be used for a load balancer. We show that it is possible to create a Othello such that the result is uniformly random.

Rebalance operation

: One useful property of the l-Othello is that although it is a memory-efficient data structure, there is a certain level of redundancy in the encoding of the query results. Consider a subset of keys $C = \{k_1, k_2, \cdots, k_t\} \subset S$, where their corresponding edges form a *connected component* in G. For all i and j values where $i = h_a(k)$, $j = h_b(k)$ for some $k \in C$, and an arbitrary l-bit integer x, execute this operation: Let $A[i] \leftarrow A[i] \oplus x$ and $B[j] \leftarrow B[j] \oplus x$. After such operation, the $\tau(k)$ value is *not* changed for any $k \in C$. This is to say that we are able to *modify the values in A and B, while not changing the* $\tau(k)$ *values* for any $k \in S$. In addition, some $A[i]$ and $B[j]$ elements are not associated with any $k \in S$ (we call them isolated elements), and we are able to modify them without changing $\tau(k)$ values for $k \in S$.

Hence, we are able to adjust the distribution of $\tau(k')$ for arbitrary queries. Direct application of this approach includes, tuning the distribution of the Othello query results in SDLB (Chapter 4.3) and for identifying alien queries in SeqOthello (Chapter 6.4.2). We are able to *rebalance* the p_t ratios by assigning all isolated elements in A and B as some random values, and execute the above operation for all connected components in G with arbitrary random x values. We call this a *rebalance* operation on the Othello. The actual distribution of p_r values depends on the actual pseudo random number generator that used in such operation. We can use different random number generators to get different $\tau(k')$ distributions. For example, when uniform random l-bit integers are used in such operation, it generates a distribution as if $\tau(k')$ is a uniform random l-bit integer. This is shown in Figure 2.5, where we compute $\tau(k')$ values for $100M$ random keys on a 12-Othello that

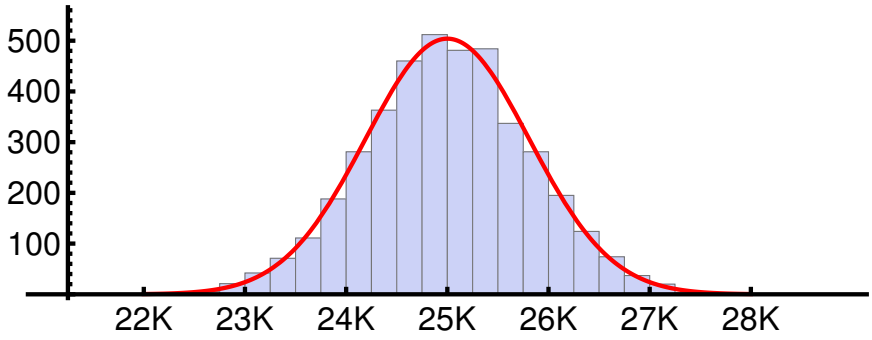


Figure 2.5: Histogram of $\tau(k)$ occurrence frequency for $100M$ random queries and 2^{12} possible values. Curve: normal distribution with parameters $\mu = 25K$ and $\sigma = 830$.

is "rebalanced" using random 12-bit integers. The curve shows a normal distribution with parameters $\mu = 25$K and $\sigma = 830$. The very small standard deviation σ suggests $\tau(k')$ can be treated as a random number, although the σ value is larger than the theoretical value.

We believe such difference results from the fact that the hash functions we implemented on a practical machine, namely h_a and h_b, are not "uniform random", since such hash functions can not be easily implemented on a practical machine. As a comparison, if we use a constant l-bit value in such "rebalance" operation, we can expect that $\Pr[\tau(k') = 0]$ to be much higher as proved in Theorem 5.

2.6 Implementation and Evaluation

In this section we show the performance for the Othello Hashing data strucutre.

2.6.1 Evaluation environment and settings

We implement the Othello query and control structures, running on different cores of a desktop computer. The memory-mode experiments are used to compare the performance of the algorithms and data structures. They demonstrate the maximum lookup speed that Othello Hashing is able to achieve on a computing device by eliminating the I/O overhead.

Platform

In the following section, unless specified otherwise, we evaluate the performance with 4 parallel query threads. The number of action is set to 256 ($l = 8$). We conduct all experiments on a commodity desktop computer equipped with one Core i7-4770 CPU (4 physical cores @ 3.4 GHz, 8 MB L3 Cache shared by 8 logical cores) and 16 GB memory (Dual channel DDR3 1600MHz).

LFSR key generator

In the experiments, a series of queries with different keys were generated and performend on the data structure. One straightforward approach is to feed the data structure with a publicly available traffic trace, or to generate a series of queries with some random number generators and store the query series in the physical memory of the machine. However, the time for transmitting the data from the physical memory to the cache is too large compared to the time used to conduct a query on the data structure. Similarly, it is not feasible to query the data structure with keys directly generated by a random number generator since the overhead for generating psedo random numbers is too high. Hence, to conduct more accurate measurement, we use a linear feedback shift register (LFSR) to generate the keys. One LFSR generates about 200M keys per second on our platform.

In fact, LFSR gives no favo because the keys are generated in a round-robin scenario, which provides the minimum cache hit ratio. LFSR traffic is actually the *worst* sequence of queries for Othello Hashing. On the contrary, in denial-of-service attack traffic, the queries concentrate on one or few keys, and they always hit the cache. Hence, the query throughput of Othello in DoS attack traffic may be higher than the value measured with LFSR traffic. We believe the result measured in LFSR traffic reflects the true performance of Othello.

Methodology

We compare Othello Hashing with three approaches for classifcation key-value query: (1) Cuckoo hashing [54] (used in CuckooSwitch [53] and ScaleBricks [55]), (2) BUFFALO [10], (3) Orthogonal Bloom filters, and (4) Bloomier Filter. CuckooSwitch [54] is optimized for a specific platform with 16 cores and 40 MBs of cache. ScaleBricks [55] is designed for a high performance server cluster. We were not able to repeat their experiments on commodity desktop computers. Instead, we compare Othello Hashing with `(2,4)`-Cuckoo tables, which is the high performance hash table used in ScaleBricks, by reusing the code from the public repository of CuckooSwitch. BUFFALO does not always return correct forwarding actions. The false positive rate is set to at most 0.01%. We also implement a technique called Orthogonal Bloom filters (OBFs) for comparison. It uses a Bloom filter to replace an Othello for classification of two sets X and Y: all keys in X hit the Bloom filter. The false positive rate is also set to at most 0.01%. For Bloomier filter, we use version that use three hash functions to determine the neighborhood of keys, and we set the ratio of hash buckets to elements to 1.23. This set of parameter is used for maximum optimization for memory space specified in [44].

2.6.2 Performance metrics

We use the following performance metrics to measure the performance of different data structures.

- **Query Structure performance metrics** characterize the performance of the query structure.
- **Query throughput:** the number of queries that a data structure is able to process per second.
- **Query throughput under update:** the query throughput measured when the data structure is being updated. It reflects the effectiveness of the concurrency control

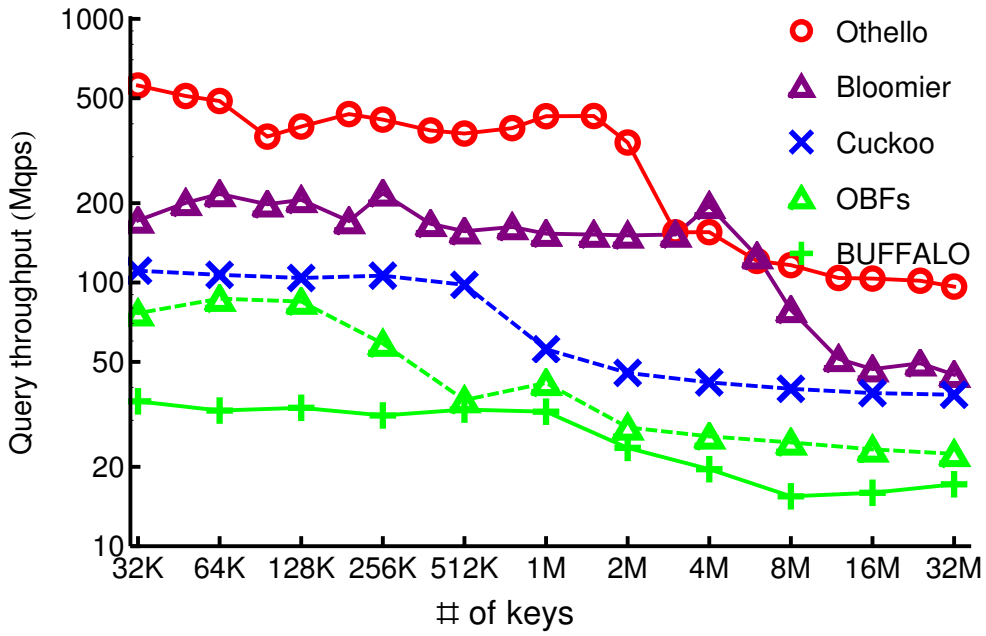


Figure 2.6: Query throughput versus number of keys.

mechanism.

Control Structure performance metrics characterize the performance of the control structure.

- **Construction time:** the time to construct a data structure.
- **Update throughput:** the number of updates that can be processed by the control structure per second. Here, an update may consist in adding a key, deleting a key, or changing the corresponding value of a key.

2.6.3 Query structure performance

Query throughput versus number of keys.

Figure 2.6 shows the query throughput of Othello, Cuckoo, BUFFALO, and OBFs. The keys are 48-bit fixed-length integer numbers, which is equivalent to MAC addresses in a typical Ethernet.

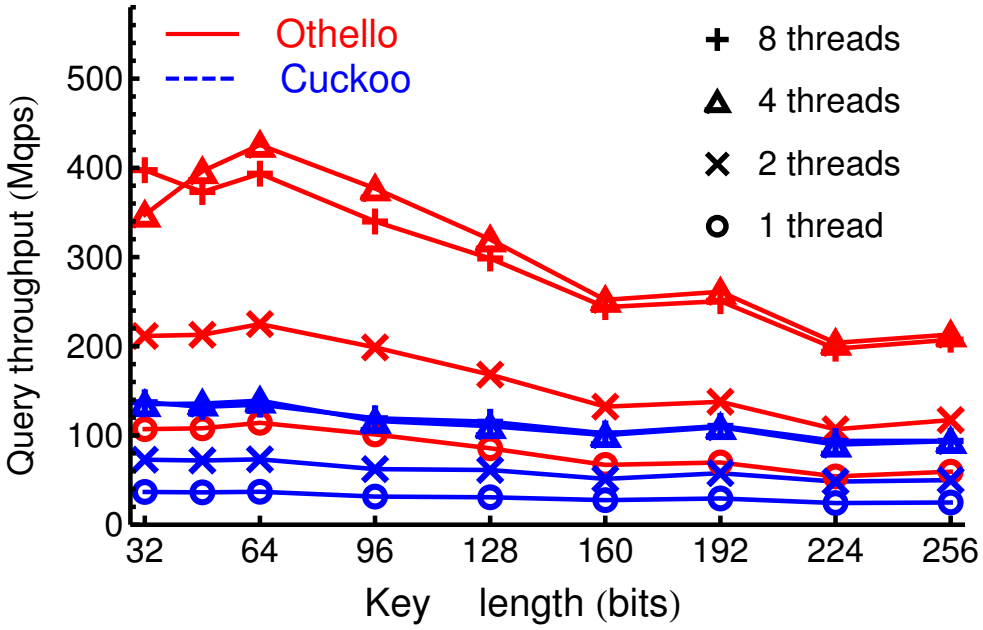


Figure 2.7: Query throughput versus key length

When n is smaller than 2 million, the throughput of Othello is very high ($>$ 400M queries per second (Mqps)). This is because the memory required by Othello is smaller than the cache size (8M for our machine). When $n \geq$ 2M, the throughput decreases but remains around 100 Mqps. This indicates that if other resources (e.g., I/O and buffer) are not the bottleneck, Othello reaches 100Mqps. The query performance decreases as the size of the query structure exceeds the CPU cache size. We observe similar results when running the evaluation on other machines with different CPUs. Bloomier filter shows lower throughput than Othello. Note that the memory space for bloomer filter is smaller than Othello and hence it may show even higher throughput when it is able to fit in the cache. Cuckoo is only about only 20% to 50% of Othello. The results of Cuckoo are consistent with those presented by the original CuckooSwitch paper[1]. Note that the measured time overhead includes that of query generation.[2]

[1]The paper [53] showed a throughput 4.2x as high as our Cuckoo results on a high-end machine with two Xeon E5-2680 CPUs (16 cores and 40MB L3 cache). It is approximately 4x as powerful as the one used in our experiments.

[2]In the evaluation of 1M keys, each query of Othello takes about 4.5 ns while generating a query takes 4.1 ns.

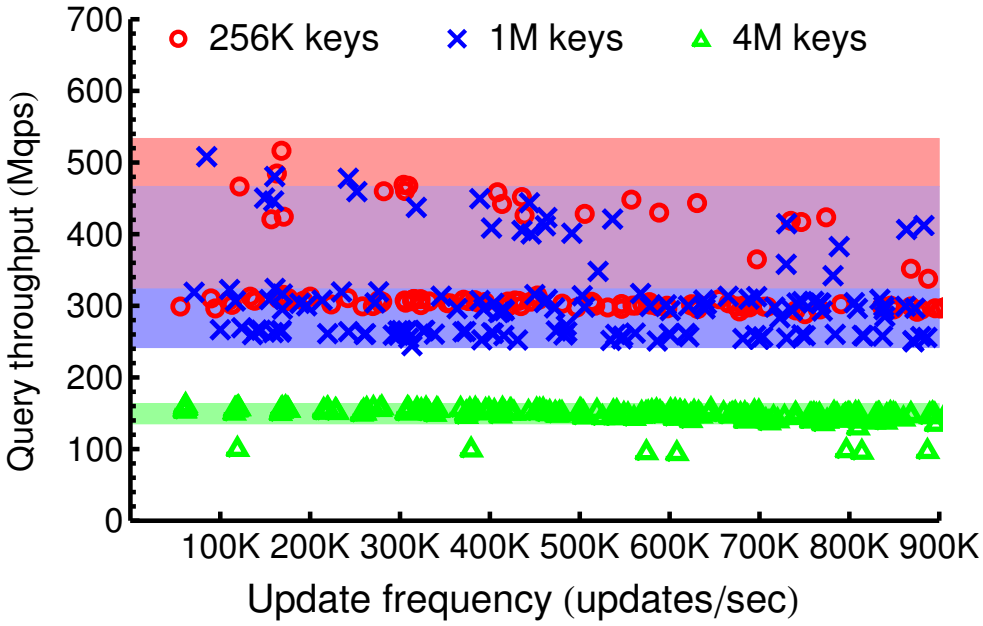


Figure 2.8: Othello query throughput under different update rates

Query throughput versus key lengths and number of CPU cores.

Figure 2.7 shows the query throughput using different key lengths. Each hashing approach contains 256K keys. As the length grows, the throughput of all types of Othello and Cuckoo decreases. Note that the memory size of Othello is independent of the key length. Hence, the throughput decrease of Othello is due to the increase of hashing time. One interesting observation is that when the length is a multiple of 64 bits, the query throughput of Othello is slightly increased. This is mainly because the experiments are conducted on a 64-bit CPU. The query throughput grows approximately linearly to the number of used threads, as long as the number of threads does not exceed the number of physical CPU cores of the platform.

Query throughput during updates.

Figure 2.8 shows the throughput of Othello during updates, including key additions, deletions, and action changes. There is only very small decrease of query throughput even when the update frequency is as high as hundreds of thousands of keys updated per second. We

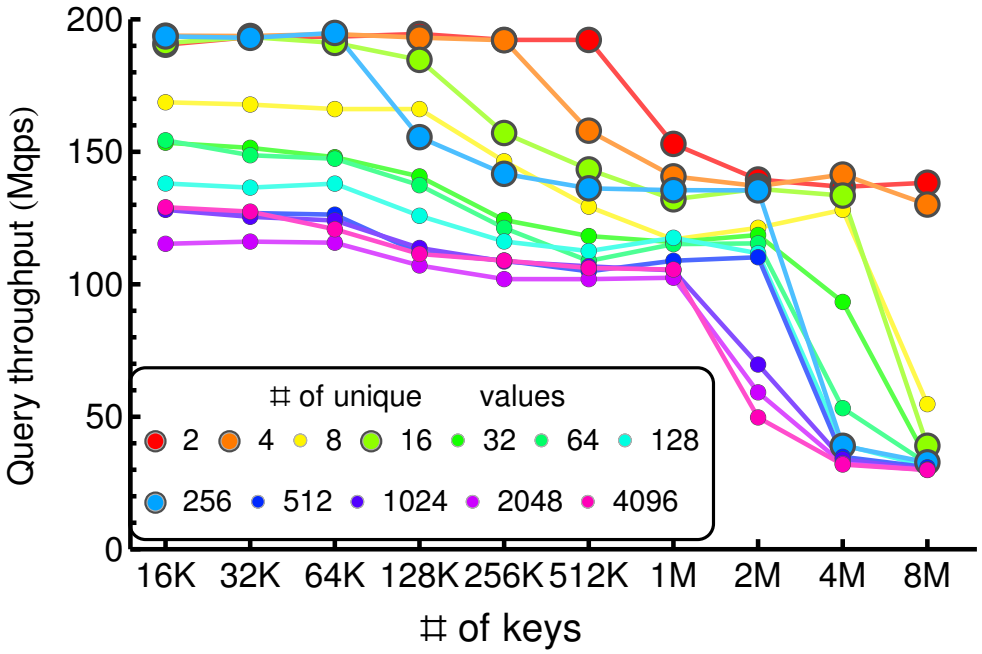


Figure 2.9: Query throughput versus number of forwarding actions

mark the one-σ (68%) confidence interval of the throughput when there is no concurrent query in Figure 2.8. Evaluation result shows that the throughput of Othello still remains in its normal range during updates. For Othello with 4M keys the throughput downgrade is negligible.

Query throughput versus number of possible output values.

Figure 2.9 shows the query throughout of using Othello for one thread for different number of forwarding actions. Using l-bit value, Othello is able to represent query result in $\{0,1,\cdots,2^l-1\}$. The throughput is better when the number of forwarding actions equals to 2, 4, 16 or 256. This is because the memory of Othello query structure is better aligned when $l \in \{1,2,4,8\}$.

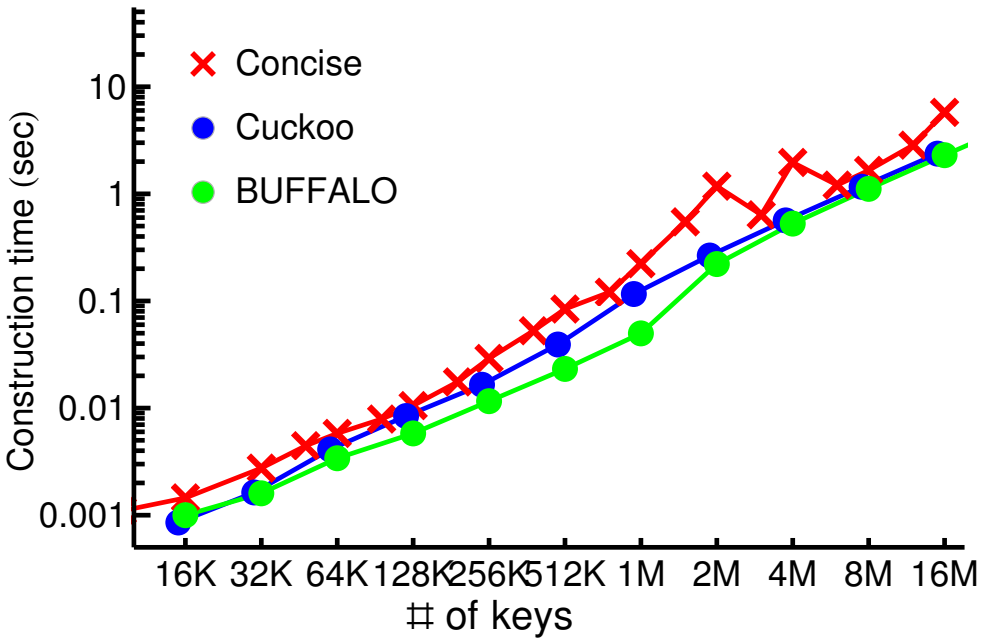


Figure 2.10: Construction time comparison among three data structures

2.6.4 Control structure performance

Construction time.

Figure 2.10 shows the average time to construct the query and control structures for one Othello with various number of keys. The construction time of Othello grows approximately linearly to the number of addresses. Although the time of Othello is larger than that of Cuckoo and BUFFALO, it is still very small. For 4M keys, it takes only 1 second to construct the Othello.

Update speed.

The update speed indicates the ability to react to network dynamics. All types of network dynamics, including host and link changes, are reflected as key additions, deletions, and action changes in the FIBs. Figure 2.11 shows the update speed of Othello in number of updates processed per second. We vary the number of keys before update and measure the time used to insert a number of new keys. Each run of the experiment is shown as a point

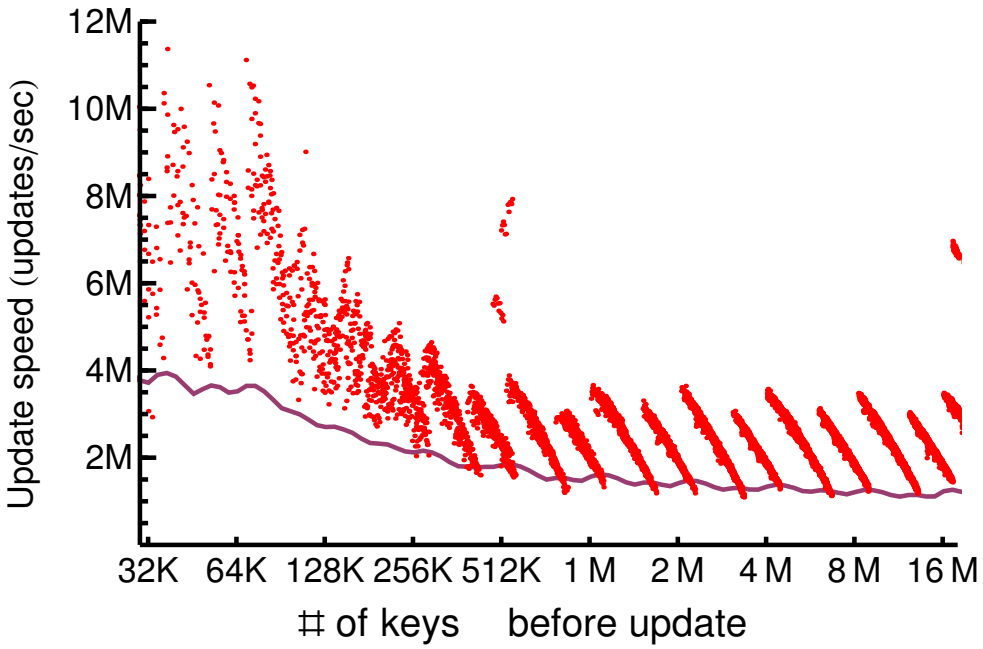


Figure 2.11: Update speed. Line: average speed including Othello reconstruction.

in the figure. Note that in a minor fraction of cases the update may cause a reconstruction of Othello, the time overhead of which is comparable to the construction time shown in Figure 2.10. The amortized speed for Othello update is shown in the curve in Figure 2.11. In most cases, it reaches at least 1M updates per second, which is sufficient for very large networks.

Remark on Othello reconstruction.

In some rare cases, adding a new key may require reconstruction of the Othello when it introduces a new cycle in to the bipartite graph. This may take non-negligible time (0.2 seconds when there are 1M keys). Theoretical results show this happens with probability less than $\frac{1.5}{n}$. This value is even smaller in practice (about 1.3 parts per million when there are 1M keys). Note that, Othello reconstruction may happen only when there is a new key added to the network. *Modifying a forwarding action of a existing key (or removing a key) never results in Othello reconstruction.* The line in Figure 2.11 shows the average update speed (including the time overhead for reconstruction). Othello reconstruction only

imposes minor impact on the update speed.

2.7 Summary of Othello Properties

Othello is a hashing algorithm that designed to support ultra-fast and memory-efficient key-value lookups. The Othello data structure follows the polymorphic data structure model. The Othello data structure can be decomposed into a query structure and a control structure. For n keys and values of l-bit integers, the query structure uses $\leq 4ln$ bits. Every query takes a small constant time including computing two hash values and two memory accesses. The control structure uses $O(n)$ bits. The expect time complexity is $O(n)$ for construction and $O(1)$ for key addition, deletion, and alterations.

Using Othello hashing, combined with domain expertise in multiple areas, we built applications in multiple areas, which is presented in the reset of this work.

Chapter 3. The Concise Forwarding Information Base

In this chapter, we present the Concise Forwarding Information Base. Section 3.1 introduces the background and the challenging issues of Forwarding Information Base. Section 3.2 presents related work. We present the the system design of Concise in Section 3.3. We then present the system implementation and experimental results in Section 3.4. Section 3.5 discusses a few related issues. Finally, we conclude this work in Section 3.6.

3.1 Background

One consensus of most recent new network design is the separation of network identifiers and locators [8]. Instead of IP, flat-name or namespace-neutral architectures have been proposed to provide persistent network identifers. A flat or location-independent namespace has no inherent structure and hence imposes no restrictions to referenced elements [9]. The Salter's taxonomy of network elements [8] is one of the early proposals that suggest the separation of network identifiers and locators. We summarize an (incomplete) list of reasons for using flat or location-independent names in proposed network architectures:

- To simplify network management, pure layer-two Ethernet is suggested to interconnect large-scale enterprise and data center networks[56–58], where MAC addresses are identifiers.

- Software Defined Networking (SDN) uses matching of multiple fields in packet header space to perform fine-grained per-flow control. Flow IDs can also be considered names, though they are not fully flat.

- Flat network identifiers have been suggested by various works to support host mobility and multi-homing, including HIP [59], Layered Naming Architecture [9], and Mobility-First [60].
- AIP [61] applies flexible addressing to ensure trustworthy communication.
- The core network of Long-Term Evolution (LTE) needs to forward downstream traffic according to the Tunnel End Point Identifier (TEID) of the flows [55].

The most critical problem caused by location-independent names is *Forwarding Information Base (FIB) explosion*. An FIB is a data structure, typically a table, that is used to determine the proper forwarding actions for packets, at the data plane of a forwarding device (e.g, switch or router). Forwarding actions include sending a packet to a particular outgoing interface and dropping the packet. Determining proper forwarding actions of the names in a FIB is called name switching. Unlike IP addresses, location-independent names are difficult to aggregate due to the lack of hierarchy and semantics. The increasing population of network hosts results in huge FIBs and their continuing fast growth.

On the other hand, the increasing line speed requires the capability of fast forwarding. To support multiple 10Gb Ethernet links, a FIB may need to perform hundreds of millions of lookups per second. Existing high-end switch fabrics use fast memory, such as TCAM or SRAM, to support intensive FIB query requests. However, as discussed in many studies [10–12], fast memory is expensive, power-hungry, and hence very limited on forwarding devices. Therefore, achieving *fast queries* with *memory-efficient* FIBs is crucial for the new network architectures that rely on *location-independent names*. If FIBs are small and increase very little with network size, network operators can use relatively inexpensive switches to build large networks and do not need frequent switch upgrades when the network grows. Hence, the cost of network construction and maintenance can be significantly reduced. For software switches, small FIBs are also important to fit into fast memory such as cache.

In this chapter, we present a new FIB design called Concise. Built with Othello, Concise has the following properties.

1. Compared to existing FIB designs for name switching, Concise supports *much faster name lookup* using *significantly smaller memory*, shown by both theoretical analysis and empirical studies.

2. Concise can be efficiently updated to reflect network dynamics. A single CPU core is able to perform millions of network updates per second. Concise makes the control plane highly scalable.

3. Concise guarantees to return the correct forwarding actions for valid names. It is *not* probabilistic like those using Bloom filters [10, 62].

Othello Hashing and Concise FIB support fast query and update (addition/deletion of names). In the resource-limited switches (data plane), Concise only includes the query component and is optimized for memory efficiency and query speed. The construction and update components are moved to the resource-rich control plane. Concise is constructed and updated in the control plane and transmitted to the data plane via a standard API such as OpenFlow. It is the first work to implement minimal perfect hashing schemes to network applications with update functionalities. Concise is designed for flat-name lookups. It does *not* support layer-3 longest prefix matching of IP addresses.

Concise is **a portable solution**, and it can be used in either software or hardware switches. We have implemented Concise in three different computing environments: memory mode, CLICK Modular Router [63], and Intel Data Plane Developement Kit [64]. The experiments conducted on an ordinary commodity desktop computer show that Concise uses only few MBs of memory to support hundreds of millions lookups per second, when there are millions of names.

3.2 Related Work

Location-independent network names. Separating network location from identity has been proposed and kept repeating for over two decades. Numerous network architectures appear in the literature that suggest this concept. As discussed in Section 3.1, a number of new network architectures adopt location-independent names. A location-independent name can be a MAC address, a tuple consisting of several packet header fields [65], a file name [66, 67], a TEID [55], etc. To route packets for flat names, ROFL [68] and Disco [69] propose to use compact routing to achieve scalability and low routing stretch. ROME [70] is a routing protocol for layer-two networks that uses greedy routing whose routing table size is independent of network size. Concise is a forwarding structure and does not deal with routing.

FIB scalability. We name some techniques used for FIBs and compare them in Table 3.1.

Hashing is a typical approach to reduce the memory cost of FIBs for name-based switching. CuckooSwitch [53] uses carefully revised Cuckoo hash tables [54] to reach desirable performance on specific high-end hardware platforms. ScaleBricks [55] also makes use of a memory-efficient data structure *SetSep* to partition a FIB to different nodes in a cluster, it does not store the names as well. We provide a comprehensive comparison of Cuckoo hashing, and Concise in Section 3.5.2. The use of Bloom filters has been proposed in some designs such as BUFFALO [10, 62]. However, they may forward packets incorrectly due to the false positives in Bloom filters, causing forwarding loops and bandwidth waste. For IP lookups, SAIL [72] and Poptire [73] demonstrate desirable throughput for IPv4 FIB queries. These solutions are usually based on hierarchical tree structures, and their performance are challenged by FIBs with large number of flat names. The Tuple Space Search algorithm (TSS) [71] is widely used for name matching with multiple files, such as in OpenVswitch and PISCES [74]. It is not designed for flat-name switching. Other solutions use hardware to accelerate name switching. For example, Wang *et al.* [75] uses

Table 3.1: Comparison among FIBs. n: # of names. L: length of names. w: # of possible actions.

In practice, Concise achieves 7% to 40% memory and >2x speed compared to Cuckoo, though they share the same order of big O time complexity.

FIB	Construction Time	Query Structure Size (bits)	Query Time	Note
Concise	$O(n)$	$\leq 4n\log w$	$O(1)$	Exact 2 memory reads per query.
(2,4)-Cuckoo [53]	$O(n)$	$\sim 1.1n(L+\log w)$	$O(1)$	Up to 8 memory reads per query.
SetSep [55]	$O(n\log w)$	$(2+1.5\log w)n$	$O(\log w)$	No method for updates. Not designed as FIB in [55].
BUFFALO [10]	$O(nt)$	αwn	$O(tw)$	Probabilistic results. Error ratio affected by t and α.
TSS [71]	$O(n(t+\log w))$	$O(n(t+\log w))$	$O(t)$	Designed for names with t fields. $t = O(L)$.

GPU to accelerate name lookup in Named Data Networks. A recent work utilize Bloom filters for set queries, which cannot be applied to our situation [76].

3.3 System Design of Concise

3.3.1 Design Overview

Consider a network of n hosts identified by unique names. The hosts are connected by SDN-enabled switches. A logically central controller is responsible of deciding the routing paths of packets. Each switch includes a FIB. The controller communicates with each switch to install and update the FIB.

Each packet header includes the name of the destination host, denoted as k. Upon receiving a packet, the switch decides the forwarding action of the packet, such as forward to a port or drop. We assume the controller knows the set S of all names in the network. In addition, Concise only accepts queries of valid names, i.e., $k \in S$. We assume that firewalls or similar network functions are installed at ingress switches to filter packets whose destination names do not exist. More discussion about eliminating invalid names is presented in Section 3.5.1.

Concise makes use of a data structure named Othello. Othello exists in both the switches (data plane) and the controller (control plane). It has two different structures in the data plane and control plane:

- **Othello query structure** implemented in a switch is the FIB. It only performs name queries. The memory efficiency and query speed is optimized and the update component is removed.

- **Othello control structure** implemented in the controller maintains the FIB as well as other information used for FIB construction and updates, such as the routing information base (RIB).

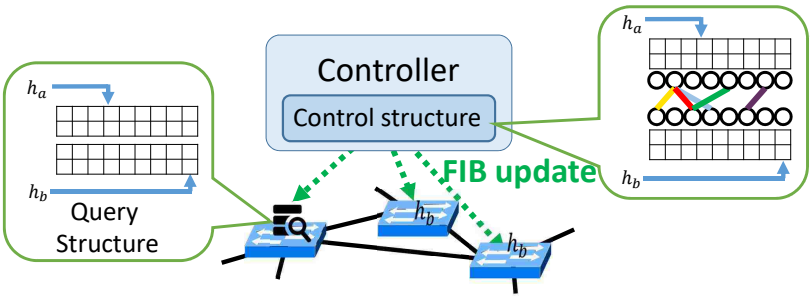


Figure 3.1: Network Overview of Concise

Upon network dynamics, the control structure computes the updated FIBs of the affected switches. The modification is then sent from the controller to each switch.

Separating the query and control structures is a perfect match to the programmable networks such as SDN. We call this new data structure design as a **Polymorphic Data Structure** (PDS). PDS is the key reason that we can apply minimal perfect hashing techniques in programable networks. PDS differs from the current SDN model. SDN separates the RIB and FIB to the control and data plane respectively. We further move part of the FIB to the control plane to minimize the data plane resource cost.

3.3.2 FIB Update and Concurrency Control

We assume that there is one logically centralized controller in the network. Upon network dynamics, the controller computes the Othellos for a number of switches and update the query structures in the switches by FIB update messages using a standard SDN API. If m, h_a, h_b do not change during the update, an update message only contains a list of elements to be modified in A and B. Otherwise, it contains the full query structure of l-Othello $\langle m, h_a, h_b, A, B \rangle$.

After receiving a FIB update message, a Concise switch modifies its Othello query structure. Instead of locks, Concise uses simple bit vectors to prevent read-write conflicts in the query structure. Experimental results show that the concurrency control mechanism has a negligible impact on the network performance.

While each Othello query is computed using two elements in A and B, there is a chance of a read-write conflict during the update. In Concise, the `query` always returns correct result. Such concurrency issue is addressed as follows.

Concurrency requirements. Let A, B be the two vectors of the query structure before an update and A', B' be the ones after the update. For a name k that exists in the FIB before and after the update, suppose $i = h_a(k)$ and $j = h_b(k)$. Both $A[i] \oplus B[j]$ and $A'[i] \oplus B'[j]$ are considered as correct actions, although they may be different. Note that, when $A[i] = A'[i]$, the values $A'[i] \oplus B[j]$ and $A[i] \oplus B'[j]$ are both correct query results, no matter how read/write events are ordered. Inconsistency only happens when both $A[i]$ and $B[j]$ are changed during the update.

Concurrency control design.

Concise observes whether the vector A is being modified. For a query for name k, if an update that affects $A[i]$ is being executed, Concise does not execute the query until the update finishes. Concise maintains two bit vectors D_1 and D_2 for concurrency control. All bits in D_1 and D_2 are set to 0 during the initialization. Each index i ($0 \leq i < m$) corresponds to an index $p(i)$ in D_1 and D_2. The lengths of D_1 and D_2 are set to 512 bits and $p(i) = i \mod 512$.

Update procedure. A pseudocode of the update procedure is described in Algorithm 2. Before an update of the Othello that will change some elements of A, Concise flips the corresponding bits in D_1, i.e., change 0s to 1s and 1s to 0s. After the update, it flips the bits with same indexes in D_2. For any index i, when Concise observes $D_1[p(i)] \neq D_2[p(i)]$, there must be no ongoing update that affects $A[i]$. Note that even if a bit index corresponds to multiple elements that are changed in an update, the bit is only flipped once.

Query procedure. A pseudocode of the query procedure is described in Algorithm 3 The query procedure for name k includes the following three steps. (1) Fetch the bit $\delta_2 = D_2[p(i)]$. (2) Fetch the value of $A[i]$ and $B[j]$. (3) Fetch $\delta_1 = D_1[p(i)]$. If $\delta_2 = \delta_1$, compute $A[i] \oplus B[j]$ and return it as the query result. Otherwise, $\delta_2 \neq \delta_1$ and we know that

Data: New value at some indexes in A and B: $A[i_1], A[i_2], \cdots, B[j_1], B[j_2], \cdots$.
Result: Updated Concise query structure

```
Affected ← ∅;
foreach i ∈ {i_1, i_2, ···} do
    Affected ← Affected ∪ {i mod 512}
end
foreach i ∈ Affected do
    D_1[i] ← 1 ⊕ D_1[i]
end
// reorder barrier
Update A[i_1], A[i_2], ···, B[j_1], B[j_2], ··· ;
// reorder barrier
foreach i ∈ Affected do
    D_2[i] ← 1 ⊕ D_2[i]
end
```

Algorithm 2: Update procedure for Concise

the Othello is currently being updated and the update affects $A[i]$. The query for k will stop and is put in a later place of the query event queue. Concise uses reordering barrier instructions to ensure the execution order in both update and query procedures.

Here, the order of flipping $D_1[p(i)]$ and $D_2[p(i)]$ during an update and the order of getting their values during a query are different. Any updates that affect $A[i]$ and start during a query must result in $\delta_2 \neq \delta_1$.

The above procedures of update and query should be executed in the given explicit order. This can be specified by compiler reorder barriers on strong memory model platforms such as x86_64, or fence instructions on weak memory model platforms such as ARM.

3.4 Implementation and Evaluation

We implement Concise on three platforms and conduct extensive experiments to evaluate its performance.

3.4.1 Implementation Platforms

- **Click Modular Router**[63] is an architecture for building configurable routers. We implement an Concise prototype on Click. It is able to serve as a real switch that

Data: Concise query structure and name k
Result: Query result $\tau(k)$

```
1  i ← h_a(k);
2  j ← h_b(k);
3  p ← i  mod 512;
4  while true do
5  |   δ_2 ← D_2[p];
6  |   // reorder barrier
7  |   α ← A[i];
8  |   β ← B[j];
9  |   // reorder barrier
10 |   δ_1 ← D_1[p];
11 |   if δ_2 = δ_1 then
12 |   |   return α ⊕ β
   |   end
   end
```

Algorithm 3: Query procedure on Concise

forwards data packets.

- **Intel Data Plane Development Kit (DPDK)** [64] is widely used in fast data plane designs[55, 77]. We use a virtualized environment to squeeze both the traffic generator and the forwarding engine on a same physical machine. This prototype is able to serve as a real switch that forwards data packets.

3.4.2 Data plane memory efficiency and MCQ

Table 3.2 shows the size of memory of different types of FIBs. We compute the memory cost used by Othello, Cuckoo hash table, BUFFALO, and OBFs, for five types of names with different sizes: MAC addresses, IPv4, IPv6, OpenFlow matching fields, and file names. Here, IP addresses are only used as examples of a name type. These FIBs are not designed for IP prefix matching. The number of actions for OpenFlow could be very large. We let the number of actions be 256 and 32,768 and compute the FIB size respectively. For the Cuckoo hash table, we use the `(2,4)` setting. For BUFFALO, we assume the names are evenly distributed among the actions, which gives an advantage to it. We use the setting $k_{max} = 8$. These settings are all as described or recommended in the original

FIB Example			**Concise**		Cuckoo		BUFFALO		OBFs	
Name Type	# Names	# Actions	Mem	MCQ	Mem	MCQ	Mem	MCQ	Mem	MCQ
MAC (48 bits)	7×10^5	16	1M	2	5.62M	2	2.64M	8	7.36M	15
MAC (48 bits)	5×10^6	256	16M	2	40.15M	2	27.70M	8	112.06M	16
MAC (48 bits)	3×10^7	256	96M	2	321.23M	2	166.23M	8	672.34M	16
IPv4 (32 bits)	1×10^6	16	1.5M	2	4.27M	2	3.77M	8	10.52M	15
IPv6 (128 bits)	2×10^6	256	4M	2	34.13M	6	11.08M	8	44.82M	16
OpenFlow (356b)	3×10^5	256	1M	2	14.46M	6	1.67M	8	6.72M	16
OpenFlow (356b)	1.4×10^6	65536	8M	2	67.46M	6	18.21M	1024	66.60M	17
File name (varied)	359194	16	512K	2	19.32M	10	1.35M	8	5.47M	15

Table 3.2: Memory and query cost comparison of four FIBs and SetSep. MCQ: maximum # of cachelines transmitted per query.

	$n=3\times10^5$ 2^8 actions	$n=1.4\times10^6$ 2^{16} actions
Name addition	75.2	107.2
Action change	65.6	88.8

Table 3.3: Entropy of one update message in bits

papers [10, 53, 55].

The memory space used by Concise is significantly smaller than that of Cuckoo, BUFFALO, and OBFs. It is only determined by the number of names n and the number of actions, and is independent of the name lengths. Table 3.2 also shows the maximum number of cachelines transmitted per query (MCQ) of these FIBs. A smaller MCQ indicates fewer data transferred from the memory to the CPU, which results in better query throughput. Concise always requires exactly two memory accesses per query. The other FIBs may have larger MCQ depending on the name length and number of actions.

Network-wide shared bipartite graph. For some networks that require every switch to store all destination names such as Ethernet, the name set S is identical for all switches in the network. Hence, all switches in the network may share the same G and $\langle h_a, h_b\rangle$. Constructing and updating the FIBs in all switches only require computing G once. e.g., the phase I of the `construct` procedure (Section 2.3.1) is only executed *once* for FIBs of all switches in the network. This indicates that the construction time overhead for FIBs of multiple switches can be further reduced. Note that for a single switch, the time used for phase I is about half of the total of `construct`.

Communication overhead. We compute the entropy of the information included in update messages in Table 3.3. The update message length grows logarithmically with respect to either the number of names n or the number of actions. The communication overhead of Concise is smaller than that of most OpenFlow operations.

Processing delay.

A data plane device maintains a queue of packets when packet arrival rate exceeds the query throughput of the FIB. The processing delay reflects the ability of the data plane to

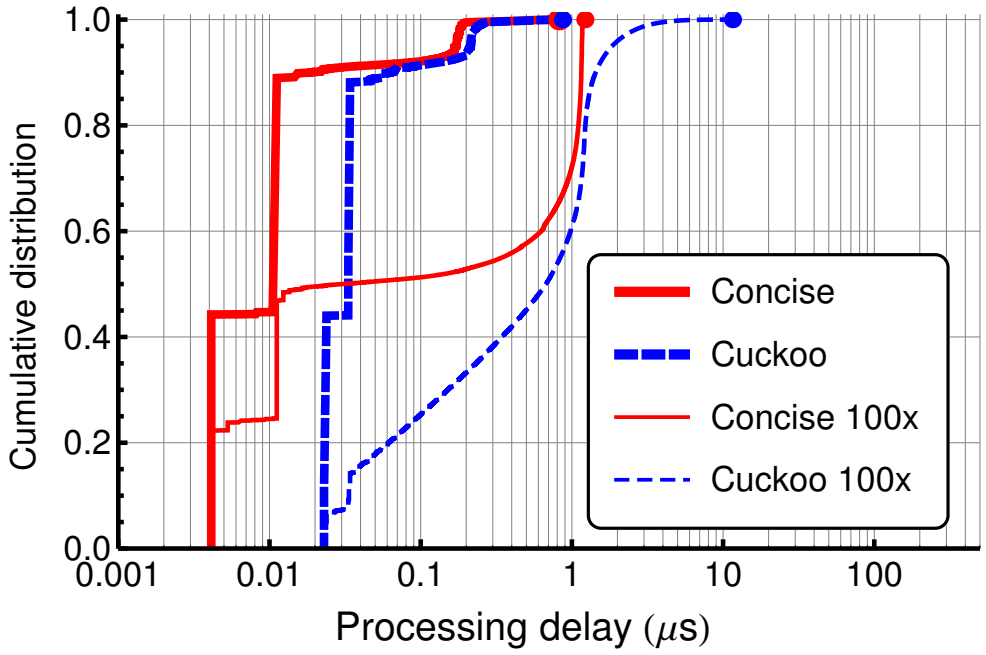


Figure 3.2: CDF of the processing delay of Concise and Cuckoo

process burst traffic. We use an event-based simulator to simulate the processing delay of both Othello and Cuckoo hashing under real traffic trace. The traffic trace is replayed in 100x speed to simulate large traffic

We conduct event-based simulations of packet processing on the data plane to study the process delay. We simulate a single-thread processor with two-level cache mechanism. The packets are processed in a first-come, first-served fashion. Each packet consists of the header and payload. The packets are put in a queue upon reception and wait to be processed by the prosessor. We measure the processing delay for real traffic data from the CAIDA Anonymized Internet Traces of December 2013 [78]. The average packet rate is about 210K packets per second. In Figure 3.2, Concise has smaller processing delay than Cuckoo before the 90th percentile, but they have similar tails. To study the processing delay under larger traffic volumes, we replay the trace 100x as fast as the original. Shown as the thin curves, the processing delay of Concise is clearly smaller than that of Cuckoo before the 60th percentile. After that, the two curves are similar, except that Cuckoo has a longer tail. Overall, the processing delay of Concise is very small ($< 1\mu$s) even under high

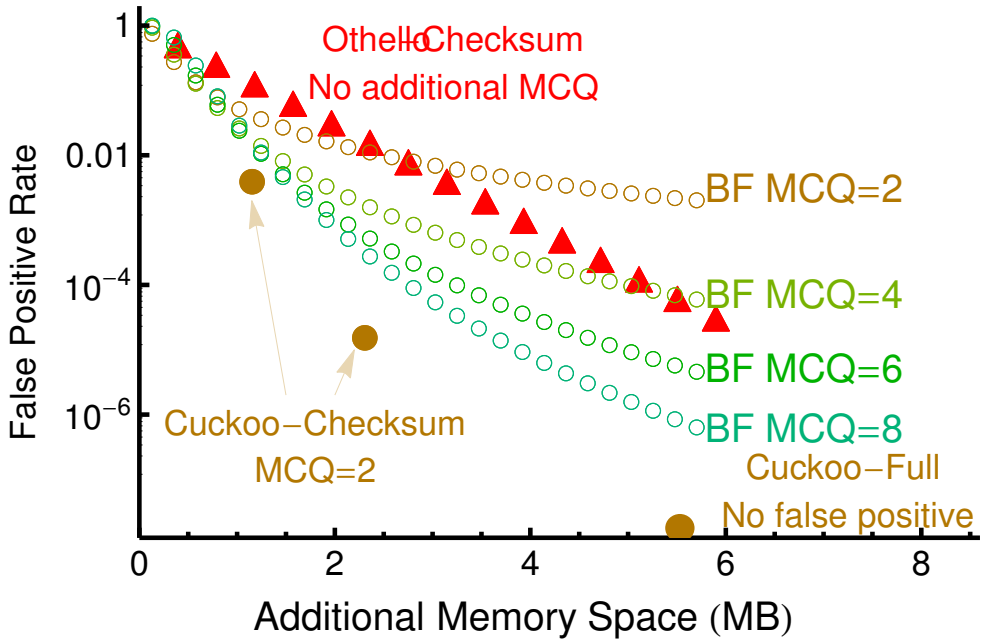


Figure 3.3: Approaches of detecting invalid names

data volumes.

Cost of detecting invalid names We also measure the cost of two approaches to detect invalid names. Figure 3.3 shows that using a 8-bit checksum (marked as Concise+Chk in the figure) has a minor impact on the query performance. We provide more analysis on the approaches in Section 3.5.1.

3.4.3 Prototype Implementation and Evaluation

Implementation on Click

We implement a Concise prototype on Click Modular Router [63]. It receives packets from one inbound port and forwards each packet to one of its 4 outbound ports. Upon receiving a packet, it queries the Othello using the address field of the packet, i.e., the name, and decides the outbound port of the packet. In addition, we implement the `(2,4)`-Cuckoo hash table, OBFs, as well as the binary search mechanism on Click. Figure 3.4 shows the forwarding throughput. The Click modules in each evaluation includes one traffic generator generating packets with valid 64-bit names, one switch that executes queries on

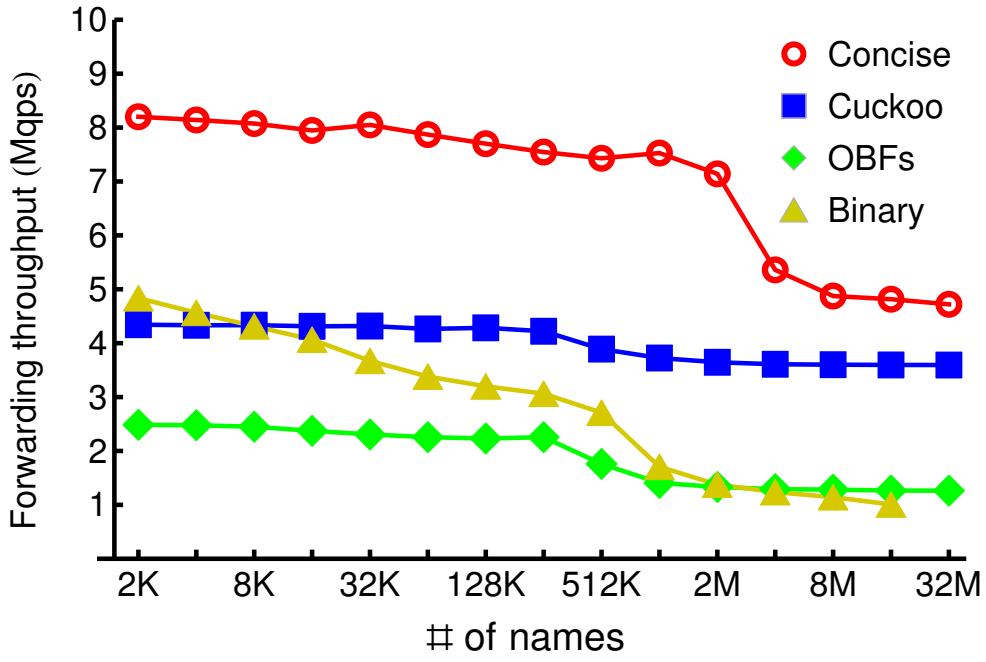


Figure 3.4: Forwarding throughput comparison on Click

the FIB, and packet counters connected to the egress ports of the switch. The experiments are conducted on one CPU core.

Results show that Concise always has the highest throughput. When $n < 2$M, Concise is smaller than the cache size and the query throughput is about 2x as fast as Cuckoo and 4x as fast as OBFs. When $n \geq 2$M, the throughput of Concise is still the highest. Meanwhile, Concise uses much less memory, about 10% to 20% of that of Cuckoo, OBFs, and Binary.

Implementation with DPDK

We also build a Concise prototype on the hardware Environment Abstraction Layer (EAL) provided by DPDK. It maintains a Othello query structure. The query structure is initialized during boot up and can be updated upon network dynamics. The prototype reads packets from the inbound ports, executes queries on the query structure, and then forwards each packet to the corresponding outbound port.

We implement both the traffic generator and FIB application on the same commodity computer using virtualization techniques. As shown in Figure 3.5, we create a guest virtual

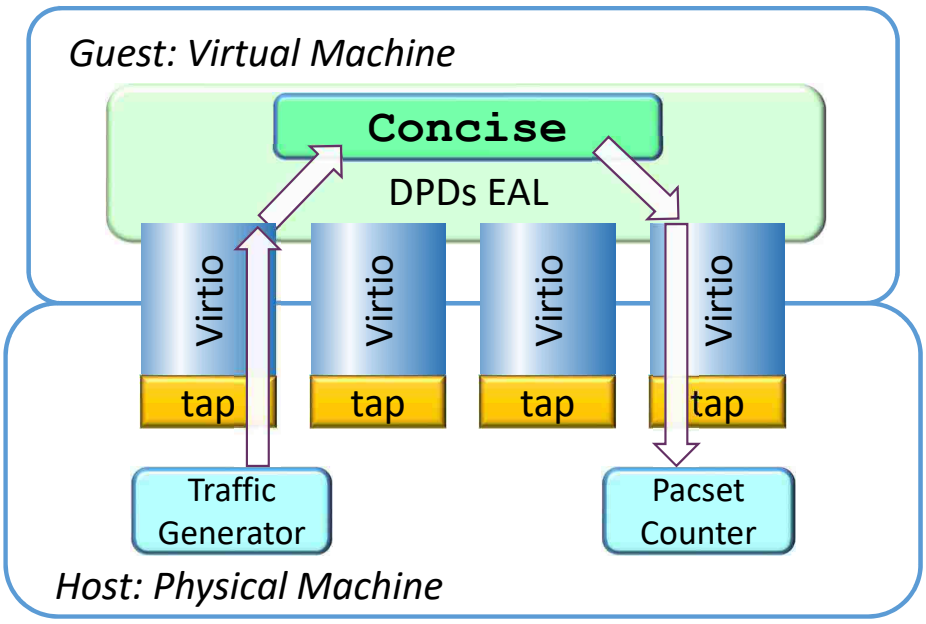


Figure 3.5: Concise prototype on DPDK

machine (VM) on the host machine using KVM and Qemu to install Concise. The VM is equipped with four virtio-based virtual network interface cards. Linux TAP kernel virtual devices are attached to the virtio devices on the host side. The programs running on the host machine communicate with the guest VM via the Linux TAPs. On the host machine, we use a traffic generator program to send raw Ethernet packets to Concise running on the VM. The host machine receives the forwarded packets from Concise and counts the number of packets using default counters provided by the Linux system.

We measure the throughput of Concise with different numbers of names. The barchart in Figure 3.6 shows that Concise is able to generate, forward, and receive more than 1M packets per second, for both 64-Byte and 1500-Byte packets. The forward throughput is at least 12 Gbps for 1500-Byte Ethernet packets. The throughput of Cuckoo is only 60% to 80% of the throughput of Concise. The forwarding throughput does not significantly change when the number of names grows

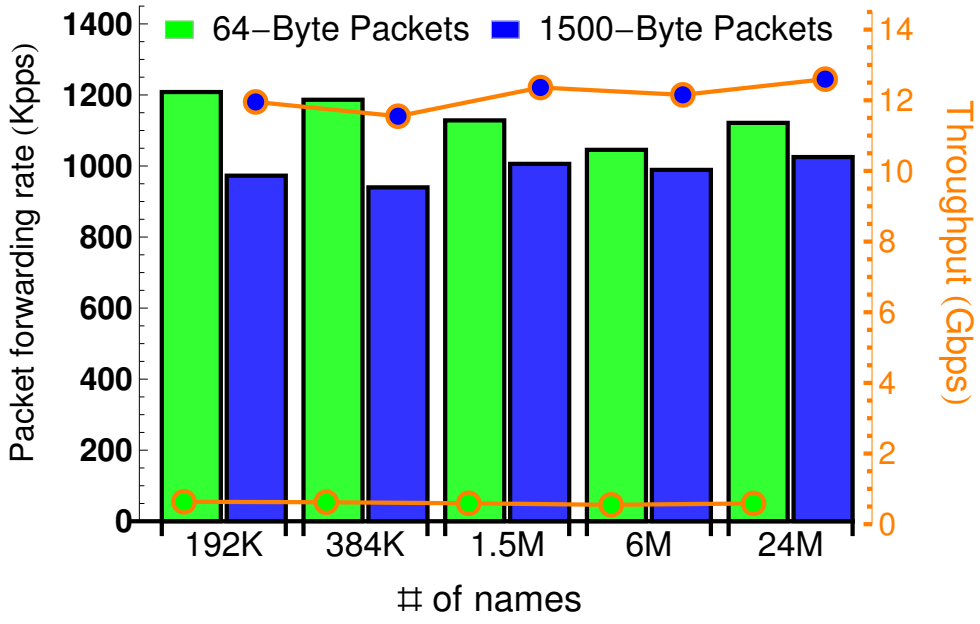


Figure 3.6: Performance of the Concise prototype on DPDK

3.5 Discussion

3.5.1 Properties of Concise for alien names

An alien name is a name that is not in S during Concise construction, such name is unexpected for name switching queries. The Othello query of alien names returns a value that is determined by the corresponding value in the memory space of the Othello query structure.

Concise is not able to distinguish alien names. This is because in Concise the *membership* information about the set of expected names is only maintained by the Othello control structure. The query structure only maintains the *classification* information of the names. In fact, a major part of the memory space for hash table based approaches (e.g, Cuckoo) is used to store copies or digests of the names in order to maintain the membership information. Meanwhile, Bloom filter [44] is widely used to maintain membership information.

The detailed discussion of the behavior of Othello alien queries is presented in [33]. In the worst case, a packet with unexpected name is forwarded to one of the neighbors of the

switch. Compared to the forwarding table miss of Ethernet, which let the packets flood to all interfaces, Concise causes no flooding.

In many large-scale flat-name networks the network addresses are all known and there are no unexpected names. However, operators may still choose one or some of the following mechanisms to enable the switches to detect the alien names.

- At an ingress switch, every incoming packet should be checked by a filter or firewall to validate that its destination does exist in the network. This filter can be implemented as a network function running on the border of the network, and can be integrated with the firewall.

- Maintain a Bloom filter at each of the switches. Packets with valid names pass this filter and are then processed by Concise FIB.

- In addition to the l-bit query results, also maintain the checksums for each name in the Concise FIBs. Adding checksums will increase the memory size of Concise. For r-bit checksums, the overall memory cost of a query structure is $2(l+r)m+O(1)$. Note that as long as $l+r$ does not exceed the word length of the computing platform, the time overhead of all operations remains unchanged.

Assuming there are in total 1M names. Fig 3.3 compares the memory and computational overheads of the above approaches. The false positive rate can be controlled to be as low as 10^{-5} with $<$ 2MB memory overhead using the filter of Cuckoo with checksums. The performance when using Bloom filters may vary depending on the parameters. We also recommend to utilize the time-to-live (TTL) value of to prevent the packet being forwarded in the network forever.

The unique property of returning an arbitrary value for an alien name may also be useful for Concise as a network load balancer: for a server-visiting flow that is new to the network, Concise can forward it to one of the servers with adjustable weights.

3.5.2 Othello versus Cuckoo and SetSep

Concise is essentially a classifier for names, and each class represents a forwarding action. Concise does not store the names. Cuckoo stores all names and actions in a key-value store.

SetSep has some properties similar to Concise. Both of them do not store names and return meaningless results for unknown names. In ScaleBricks [55], SetSep is only used as a separator to distribute the FIB to different computers, rather than the FIB. Meanwhile, the update scheme for SetSep is not explicitly explained [55], and there is no discussion about handling dynamic FIB size growth.

In addition to the memory size results in Table 1, we show some comparison results of SetSep in what follows. The construction speed of SetSep is slower than that of Concise and Cuckoo by more than an order of magnitude: 10 seconds for one single FIB of 1M names in our experiments. We also measure the update speed of SetSep without adding new names, which turns to be less than 10K/s (< 1 % of Concise). The query speed of SetSep is higher than that of Cuckoo. SetSep needs to compute $1+l$ hash values and read $2+2l$ values for each query. We implement a static SetSep with 1.4M names and $l = 8$, using 2.19MB memory. Its query throughput is 211 Mqps using 4 threads. In comparison, Concise with the same settings uses 4M memory and reaches 470 Mqps.

In addition, we summarize the reasons of the performance gain of Concise as follows. (1) Othello does *not* maintain a copy of the names in the query structure. The memory size of the query structure is much smaller than the other solutions. Concise demonstrates higher cache-hit rate, which leads to better performance on cache-based systems. (2) The query procedure does not contain any branches (e.g, `if` statements). This helps the CPU to predict and execute the instructions in the query procedure. (3) The efficient concurrency control mechanism further improves the query speed of Concise.

3.5.3 Example Use Case

Concise provides desired FIB properties for many current and future architecture designs that adopt flat names as mentioned in Section 3.1. We present a use case where it can be applied in a large enterprise network.

A large enterprise or data center network may include up to millions of end hosts and more VMs [79]. In these networks, internal flows contribute to the most bandwidth, which can be forwarded by Concise using destination names on Layer 2. The destination of a packet in this network can only be either a host or a gateway. We require hosts in the network voluntarily check the validity of the packets before sending them out. This can be easily achieved using software firewalls such as *iptables*.

As of the gateway, we require it to execute two network functions: (1) For packets going out from the network, perform Layer 3 routing using the external IP of the destination. This is a basic function a router. (2) For packets going into the network, filter out all packets with invalid destinations. This can be implemented by a firewall. The packets will be forwarded using the Layer 2 names of the destinations. In addition, we require all packets in the network to carry a time-to-live (TTL) value to prevent packets from being forwarded forever in case packets with invalid names pass the firewalls.

3.6 Summary

Concise is a portable FIB design for network name lookups, which is developed based on a new algorithm Othello Hashing. Concise minimizes the memory cost of FIBs and moves the construction and update functionalities to the SDN controller. Evaluation shows that Concise uses the smallest memory to achieve the fastest query speed among existing FIB solutions for name lookups. We expect that Othello Hashing, as a fundamental network algorithm, will be used in a large number of network systems and applications where existing tools such as Bloom Filters and Cuckoo Hashing may not be suitable.

Chapter 4. SDLB: Software Load Balancer

SDLB is a scalable and dynamic software load balancer for fog and Mobile Edge Computing (MEC). In this chapter, we first introduce the background of software load balancing in cloud systems in Section 4.1. We present the system design of SDLB in Section 4.3. We show the evaluation results in Section 4.4. We discuss possible future work in Section 4.5. and present related work in Section 4.2. Finally, we conclude this work in Section 4.6.

4.1 Background

Although mobile hardware continues to improve, it is still relatively resource-constrained compared to static computing hardware. To provide resource of computation, storage, and bandwidth to massive mobile computing devices such as those of the Internet of Things (IoT), strong back-end servers in a remote cloud is a common solution. However, many modern applications such as augmented reality (AR) and real time monitoring/control are latency-sensitive and may suffer the long round-trip delay to the cloud. Hence, Mobile Edge Computing (MEC) has been proposed to shift computing and storage capacities from the remote cloud to the network edge [13, 14]. The nodes that provide resource to mobile devices are called *MEC hosts*. Fog Computing is a computing paradigm with a similar objective of MEC [80–82], where the nodes providing resource are called Fog nodes. MEC and Fog may differ in specific characteristics. For example, most MEC hosts are deployed by the mobile service provider while Fog nodes may be network devices or terminals belonging to different users. MEC and Fog are close in their network models and most characteristics [83]. *In this work, we study with a generalized network model which can be*

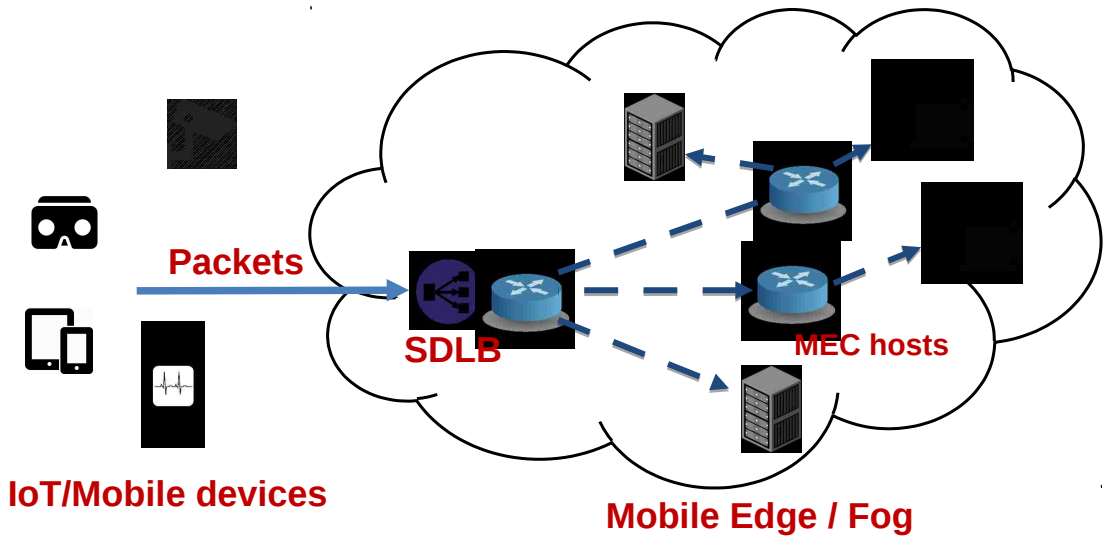


Figure 4.1: Network model

applied to both MEC and Fog Computing.

We consider a MEC network consisting of mobile devices and a number of MEC hosts as shown in Figure 4.1. The MEC hosts provide various types of resource to mobile devices, such as CPU, memory, and disk, for difference application requirements including computation, storage, and optimization. The MEC hosts communicates with a remote cloud for further processing or storage if necessary. For example, an IoT sensing device collects the environment data and transmits the data to one of the MEC hosts [14]. The MEC hosts aggregates the data, conducts initial analysis, and transmits aggregated data to the remote cloud. For example, a video analytics application running on MEC hosts may detect special events from the date reported by video cameras, such as a lost child or an intruder, and then reports these events to a control center [13]. As another example, for an AR application running on a smartphone or tablet, the MEC host is able to provide local object tracking and local AR content caching with short latency for the mobile device [13].

MEC provides *resource virtualization* to mobile devices. In fact, a mobile device has no information about which MEC host is actually providing its requested resource. An MEC network deals with the data traffic from mobile devices to MEC hosts, serving various applications and purposes. In the network layer, we classify the packets from mobile devices

to MEC hosts into two groups. 1) *Stateful packets*. A packet is stateful means one of the MEC hosts stores its state, i.e., some existing information related to the packet. The packet has to be forwarded to that particular MEC host. The state can be in the device level, application level, or flow level. For example, *device-level state* can be some data reported by the same device of this packet. *Application-level state* can be some data generated by the same application of this packet. *Flow-level state* can be the previous packets of a same flow of this packet. 2) *Stateless packets*. A stateless packet can be forwarded to any MEC host, which can be a query/write to a distributed database, a request of computation offloading, or the first packet of a flow.

The *MEC load balancer* (MEC-LB) is a network function deployed between the mobile devices and MEC hosts, which assigns a MEC host as the destination for every packet from a mobile device. A MEC-LB must serve the following two properties. **P1: For every stateful packet, the MEC-LB performs a lookup to get the designated MEC host based on the packet ID. P2: For every stateless packet, the MEC-LB selects one of the available MEC hosts according to capacity of the hosts.** Different MEC hosts are assigned with different probabilities based on their capacities.

Due to the new characteristics of MEC, existing solutions of cloud load balancers [15–17] cannot be directly applied to MEC-LB. We summarize the desired requirements of a MEC-LB with comparison to a cloud load balancer as follows.

1. *Scalable*. A MEC-LB should be fast and scalable to process massive amount of packets from a large number of mobile devices. A cloud load balancer has a similar requirement [15, 17].

2. *Software based*. Compared to the hardware load balancers in a cloud [15, 16] which is a fixed amount of extra financial cost, a MEC-LB implemented in software is more flexible and cost-efficient for MEC. It can be deployed as a virtual network function running on a MEC host or a component of a router/switch.

3. *Memory-efficient*. With small memory cost, a MEC-LB is easier to fit with fast (and

hence expensive) memory such as cache. Unlike a resource-rich cloud, which can use a server cluster for its software load balancer [17], a MEC network has limited resource to host a MEC-LB.

4. *Adaptive to MEC changes*. MEC hosts are heterogeneous [80]: each carrying different amount of physical resource available to mobile devices. Since many MEC hosts are built on existing devices or terminals deployed for other purposes [82, 84], they may join/leave the network depending on their own needs. A MEC host may run out of its capacity for its own applications and hence becomes unavailable to mobile devices. In contract, servers in a cloud are mostly homogeneous and more stable in the network.

5. *Portable*. The MEC-LD should be a portable solution which does not rely on any special hardware platform. It is because a MEC-LD may be deployed at any network device or terminal. The packet ID can be arbitrary, such as 5-tuple [85], MAC address, or any network names in new Internet proposals [60, 67].

In this chapter we present a new design of a Scalable and Dynamic Load Balancer for MEC, called SDLB, that satisfies the above requirements. SDLB is built on Othello. SDLB uses very small memory and has an important feature which is a perfect fit of a MEC load balancer: to process a packet with an ID, if the ID-host relationship is specified in SDLB, it quickly returns the host ID to which the packet should be forwarded; if no ID-host relationship is specified, it quickly returns a random host ID, and the *probability distribution of host IDs can be specified and adjustable* according to the host capacity.

Evaluation results show that SDLB is faster and uses less memory, than a widely-used load balancer design (hash table + consistent hashing) which is the core algorithm in Google's cloud load balancer Maglev [17]. SDLB is also adaptive to network dynamics.

4.2 Related Works

MEC and Fog.

The idea of shifting storage and computing from clouds to the edge of the network has been proposed by many similar concepts such as mobile edge computing (MEC) [13, 14], fog computing [80–82], edge computing [86] and mobile cloud computing (MCC) [87]. Fog and edge computing have interchangeable definitions, both of which allow heterogeneous devices on the path to the remote cloud to offer storage and computing resources. MEC on the other hand relies on servers owned by mobile providers (e.g., Cloudlet [88]) behind Radio Access Network (RAN). MCC focuses more on the federation of the cloud, proximate mobile computing entities and a plethora of mobile devices.

Software Load Balancer.

HAProxy [89] and Linux Virtual Server (LVS) [90] are the mostly used open-source software load balancers. Since HAProxy operates on Layer 7, it is able to perform complicated tasks on traffic flows such as SSL authentication and traffic regulation. The proposed SDLB and LVS focus on Layer 4. Due to its simplicity, LVS is cheap and extremely fast. Meglev [17] has been used as Google's network load balancer since 2008, which is able to achieve line-rate throughput. Compared to Meglev, LVS is not as optimized in term of performance because it is designed for portability. Maglev relies on a server cluster hence cannot be applied to MEC/Fog. In contrast, SDLB is general-purpose and portable. Duet [91] is a hybrid software and hardware load balancer with all the benefits of software load balancers as well as enjoys low latency and high throughput. Duet relies on the recently provided APIs for fine-grained control over ECMP and tunneling functionality on commercial switches. It cannot be run at general-purpose platforms.

4.3 System Design

4.3.1 System overview

The key idea of using SDN for SDLB is as follows. Since the data plane of a load balancer is resource limited, we let the data plane of SDLB only include the packet processing component and is optimized for memory efficiency and processing speed. The construction and update components of SDLB are moved to an *LB controller*, which uses more memory to keep the data plane consistent to network changes. The data plane of SDLB (called *SDLB-DP*) is able to fit to fast memory such as cache to achieve better processing speed. Since network changes are much less frequent than the incoming packet rate, the LB controller can be implemented with relatively slow memory such as RAM. Upon network dynamics, the controller computes the updated SDLB-DP and sends the modification to SDLB-DP similar to existing SDN APIs [3].

We consider the SDLB-DP as *a key-value query structure as well as a deterministic randomizer*. Every packet carries an ID and is processed by the SDLB-DP. 1) For a stateful packet, SDLB-DP takes the packet ID as the key and returns a MEC host ID as the value. The MEC host specified by the value should be the host that includes the state of this packet. Here the SDLB-DP servers as a key-value query structure. 2) For a stateless packet, SDLB-DP returns a random MEC host ID depending on the packet ID and the probability distribution of this random selection should be adjustable. Here it serves as a deterministic randomizer. In the following. we introduce the data structure that can achieve these two functions simultaneously.

4.3.2 Data structure and algorithms

SDLB-DP is in a two-level structure as shown in Figure 4.2. The first level provides a many-to-one mapping from a packet ID k to an l-bit value $\tau(k)$, using a l-Othello.

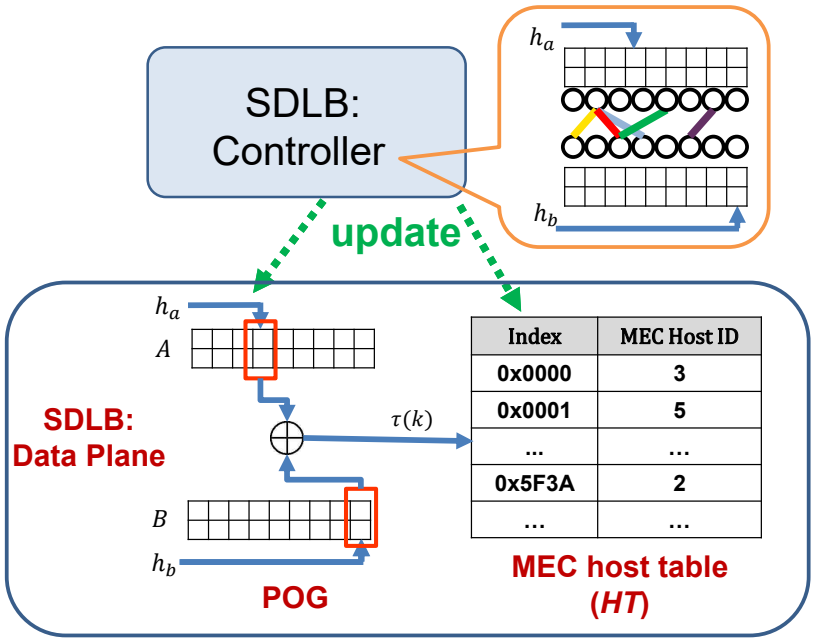


Figure 4.2: SDLB structure

First level: l-Othello

The first level of SDLB structure is an l-Othello $\boldsymbol{O}(S,T)$, where S is the set of all stateful packets. $S = \{k_1, k_2, \cdots, k_n\}$. T is the list of corresponding $\tau(k)$ results for all $k \in S$. $T = (t_1, t_2, \cdots, t_n)$. The value $t = \tau(k)$ for any $k \in S$ is used as the index on the second level structure.

Second level: MEC host table (HT)

SDLB also maintains an array in the data plane as the MEC host table (HT). HT contains 2^l elements. $l = 16$ is sufficiently big for SDLB. After computing the value $\tau(k)$ from the Othello, SDLB-DP returns the value stored in $HT[\tau(k)]$ as the MEC host ID. Since $0 \leq \tau(k) < 2^{16}$, the table HT is very small and is able to fit into fast memory.

When SDLB-DP gets a packet with key k, it returns a particular value $HT[\tau(k)]$. This value is deterministic and hence SDLB guarantees to forward all packets with a particular packet ID to the same host.

4.3.3 SDLB update

We may deal with multiple types of network dynamics including: **1)** Flow addition and leave of the network. **2)** MEC host capacity change. MEC hosts are heterogeneous and may be resource-limited. Hence the capacity of a MEC host to serve mobile packets may change frequently. **3)** MEC host join and leave. MEC hosts may be built on existing devices or terminals deployed for other purposes [82, 84]. Hence their churn rate is much higher than that of cloud servers. **4)** State migration. When a MEC host runs out of capacity or intends to leave the network, the state stored on it need to be migrated to other hosts.

In case 1), if there is arrival of new flows, SDLB-DP does not need to be updated to include the new flow information to guide the processing of the future packets of these flows. It is because the result $\tau(k)$ is deterministic for a same packet ID. However, when a MEC host creates state for a flow/application/device, it will notify the SDLB controller. The controller will maintain the packet ID to MEC host relationship according to the state.

We start to introduce update mechanisms by introducing the "merge" operation on the SDLB-DP. We assume 2^l is far greater than the number of MEC hosts. Consider any two packets k_1 and k_2 with $\tau(k_1) \neq \tau(k_2)$ but they are assigned to the same host, i.e., $HT[\tau(k_1)] = HT[\tau(k_2)]$. We can execute an update on the Othello so that $\tau'(k_2) \leftarrow \tau(k_1)$. This operation "releases" the element $HT[\tau(k_2)]$. After such operation, changing the value of $HT[\tau(k_2)]$ will not affect any stateful packets. Hence, $HT[\tau(k_2)]$ can be used as a buffer for future SDLB-DP updates.

Case 2) can be treated by modifying the HT table. To arrange more packets to a host, SDLB assigns more elements in HT to the corresponding host ID. This can be achieved by changing the values of the "released" elements in HT and it does not affect any stateful packets.

For case 3), in addition to assign the capacities, when a host leave, it may requests to migrate packets with key k to another host, which results in case 4). To modify the host assignment of a particular packet with key k, SDLB modifies the Othello value so that $\tau'(k)$

points to a "released" element in the HT and assigns $HT[\tau'(k)]$ to the new host ID.

Note that after some modifications on the Othello, the probability values P_r may change. SDLB executes the procedures described in Section 2.5.3 to rebalance the probability of possible $\tau(k')$ values for the stateless packets.

4.4 Evaluation

In this section, we evaluate the performance of the SDLB. We compare SDLB with a widely-adopted approach for network load balancer. That approach first perform a lookup in a hash table that maintains the ID to host mapping for stateful packets. If the lookup fails then the load balancer knows the packet is stateless and uses consistent hashing to assign a host for it. We measure the memory size, update speed, and data plane throughput of SDLB and the compared approach (referred as HashTable in the following). In our implementation, we use HashMap that uses a self-balancing binary search tree. In our experiments we assume the packet IDs are 64-bit values while the MEC Host IDs are 16-bit integers.

4.4.1 Memory efficiency

We compute the memory space used by SDLB-DP and the HashTable approach in Table 4.1. We show three typical types of packet IDs: (1) HashValue, in which the load balancer computes a hash value as the digest of the metadata (e.g, URL). (2) 5-tuple, it refers to the src/dst IP addresses and port numbers, and protocol number of a IP packet. (3) OpenFlow matching fields, which is a 356-bit value. SDLB-DP uses much smaller memory space than HashTable does. Note that the Othello data structure does not maintain a copy of the keys in its memory, while the HashTable must maintain a copy of the keys to identify hash conflicts. Hence, the memory space of SDLB-DP grows approximately in proportion to the number of stateful packet IDs. As a comparison, the space size of the HashTable grows with both the number of stateful IDs and the length of the keys.

Table 4.1: Memory size comparison

ID type	# stateful entries	SDLB	HashTable
HashValue (64b)	96K	640K	1.41M
5-tuple (104b)	256K	1.63M	5.25M
OpenFlow (356b)	1M	6.13M	72M

4.4.2 Update speed of SDLB

We measure the update speed of SDLB and show it in Figure 4.3. In each experiment, we randomly execute update operations on the SDLB controller. The numbers of stateful packet addition and deletion are set equal. The blue curve shows the throughput of SDLB updates. **SDLB is able to support about one million update requests per second.** We observe the update throughput varies against the number of stateful packet IDs. SDLB reaches higher throughput when the number of packets are close to values like 2^n or 3×2^n for some integer n. This is in consent with the properties of the Othello data structure discussed in Section 2.3.

We also measure the efficiency of the "rebalance" operation discussed in Section 2.5.3. The red curve in Figure 4.3 shows that the time used to rebalance the SDLB grows linearly to the number of packet IDs. It takes less than one second to rebalance when there are 2M stateful IDs.

4.4.3 Data plane throughput

We compare the data plane throughput of SDLB and HashTable. In Figure 4.4, we vary the number of stateful IDs maintained by SDLB and the HashTable and measure the query throughput of both approaches. Experimental result shows that SDLB-DP has a $>$10x higher throughput than the HashTable when the number of stateful IDs is less than 300K. It still reaches at least 4x better throughput when the number of stateful IDs grows. The throughput of both approaches decreases as there are more IDs. This is because the experiment is carried on a commodity desktop computer, and the size of both data structures

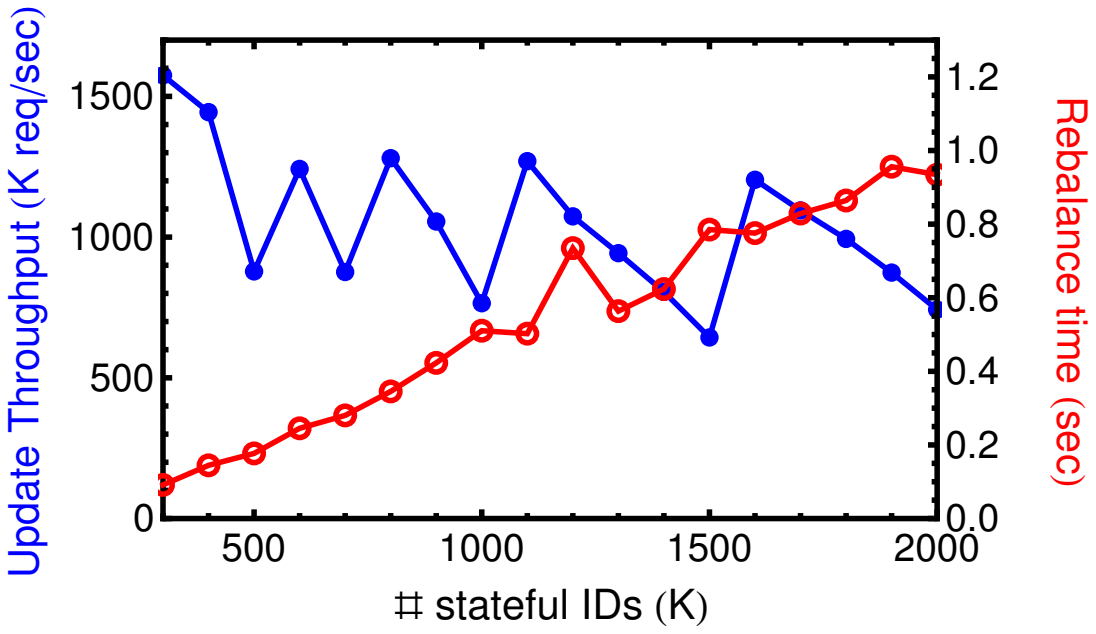


Figure 4.3: Update efficiency of SDLB

grows linearly to the number of stateful IDs. With 1000K names, SDLB is able to fit into the L3 cache of the CPU. HashTable fails to fit into the cache for 500K names. In summary, SDLB is much faster than HashTable. We also measure the data plane throughput under different types of traffic. We fix the number of stateful IDs as 600K and vary the fraction of stateful queries. Fig 4.5 shows that such fraction does *not* affect the query throughput of SDLB.

4.5 Discussion

Each MEC host needs to know its working load because of two reasons. 1) Each MEC host needs to provide provable and controllable resource/performance isolation between the original applications and mobile devices. 2) The SDLB controller needs the information to set the weight of the MEC host adaptively according to the available resources. The vanilla solution by analyzing resource consumption for all MEC traffic through existing monitoring APIs (e.g., Intel Performance Counter Monitor API [92]) is expensive and not scalable. Meanwhile, empirical studies [93–95] have shown that the network traffic is

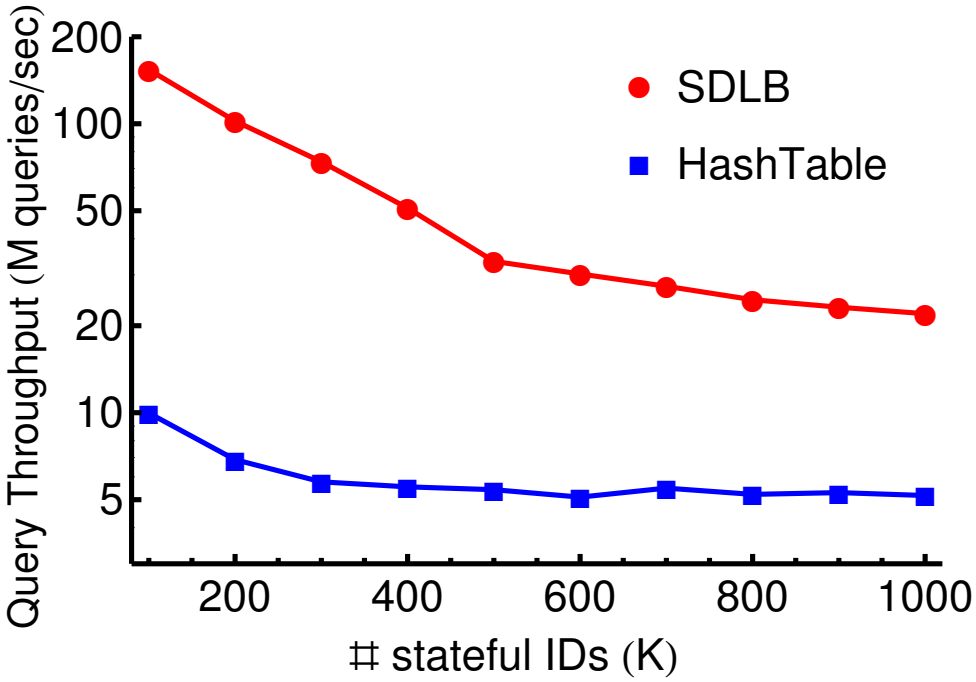


Figure 4.4: Data plane throughput vs. Number of stateful IDs.

dominated by a small fraction of *elephant flows*. We argue that tracking elephant flows on the MEC host is one cheap and efficient method for load estimation. Elephant flow detection can be achieved very efficiently. For instance, Myopia [93] leverages count-min sketch [96] to measure flow sizes for its provable tradeoff between space and accuracy of flow size estimation. If the size of a flow exceeds a threshold, an elephant flow is identified. Moreover, since the set of elephant flows substantially overlap stateful flows, the elephant flow information can be further exploited for coarse-grained tasks such as state migration in the presence of overloading.

4.6 Conclusion

We present the design of a fast and dynamic software load balancer for MEC and Fog, called SDLB. SDLB is built on a new data structure named Othello whose core algorithm is minimal perfect hashing. Experimental results show that SDLB is faster by 4x to 10x and uses less than 50% memory compared to existing solutions. In addition SDLB provides

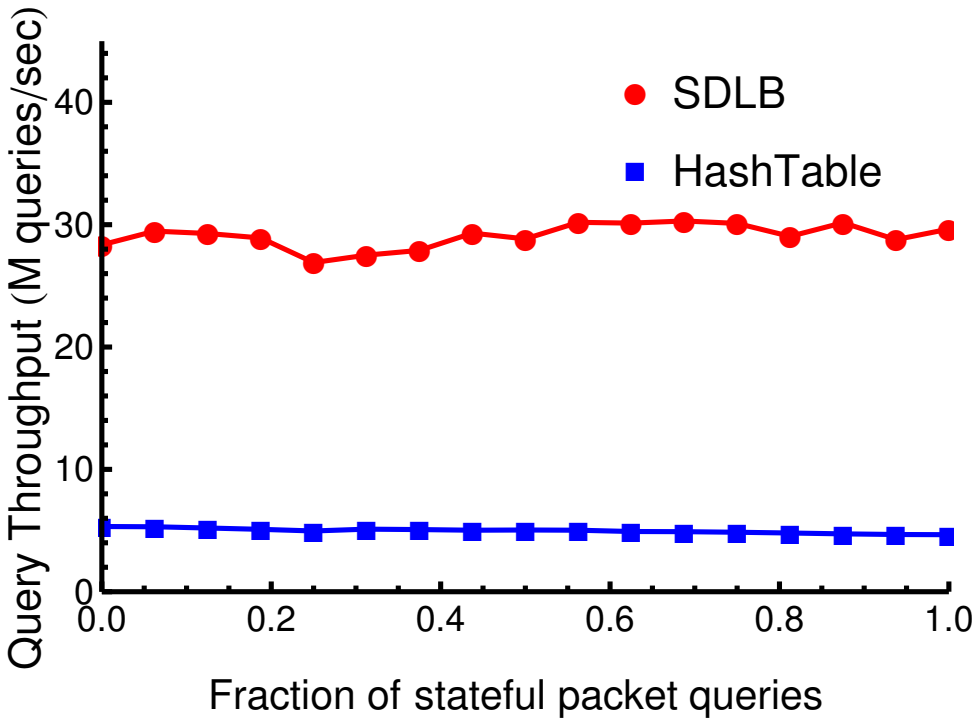


Figure 4.5: Data plane throughput vs. Fraction of stateful ID queries.

fast updates. We believe Othello has the potential to become a fast and memory-efficient solution for software-based networking in future applications.

Chapter 5. MetaOthello: Taxonomic Classification of Metagenomic Sequences

In this chapter, we present MetaOtello, a system to to support ultra-fast taxonomic classification of metagenomic sequences with k-mer signatures. We first discuss the background and motivation in Section 5.1. The system design of MetaOthello and the theoretical analysis for its properties are presented in Section 5.2. We show the evaluation results in Section 5.3.

5.1 Background and Motivation

Metagenomics is the study of genomic content obtained in bulk from an environment of interest, such as the human body [28], seawater [29], or acidic mine drainage [30]. Metagenomics studies often generate tens of millions of sequencing reads in order to capture the presence of microbial organisms and quantify their relative abundances, rendering the classification and analysis of these data a logistical challenge.

One of the major computational challenges in the analysis of metagenomic data is the classification of each sequencing read into the most-specific biological taxon to which sequence conservation supports its assignment. Specifically, a read is classified as belonging to a taxon if it has high sequence similarity with the reference genomes collected for that taxon, a process made possible by the large deposits of reference sequences collected in recent years for a variety of microbial species. In 2014 alone, more than 10,000 sequence records were newly added to the NCBI RefSeq database thanks to the accessibility of high-throughput sequencing technology.

Existing classification methods can be divided into two broad categories: alignment-

based and alignment-free. The former approach, implemented most popularly as BLAST [97], assigns each read to the taxon that affords the best alignment with its reference genomes. Several methods, including MEGAN [98], PhymmBL [99], and NBC [100], apply additional machine-learning techniques to BLAST results to increase classification accuracy. These methods are often slower than BLAST alone, rendering them computationally prohibitive for large-scale analysis of many millions of short reads. However, the recent development of Centrifuge [101] has significantly improved the scalability of alignment-based algorithm using FM-index. Besides using genomic sequences as reference, the recently published tool Kaiju [102] performs alignments towards protein sequences, achieving faster classification speed than existing tools.

The other line of work, pioneered by LMAT [103] and Kraken [104], classifies a read using exact *k*-mer matches between the read and reference sequences belonging to the target taxon, thereby avoiding inefficient base-by-base alignment while maintaining a sensitivity and specificity comparable to the alignment-based approach. This approach is generally faster than alignment-based methods and allows for greater flexibility in reference material because it requires only the collection of *k*-mers extracted from reference sequences belonging to each taxon. Thus *k*-mers extracted from DNA or RNA sequencing data can be included as reference material without being assembled, increasing the sensitivity of the algorithm in capturing natural variants that are often missed using reference genomes alone.

The above alignment-free approaches rely on the use of indexing structures for *k*-mer matching. For example, Kraken indexes its lexicographically sorted *k*-mer database using a minimizer offset array, while Clark uses a hash table to store the mapping between a *k*-mer and its classification information. Both Kraken and Clark require computers with large memory to support the construction of their indexing structure (at least 170 GB RAM) and *k*-mer querying during classification (at least 70 GB RAM). Although there are variations of both algorithms with smaller memory footprints, they often afford significantly lower

accuracy and much slower execution speed compared to the full version. For this reason, the ever-increasing amount of sequencing and reference genome data call for tools with better scalability in both memory and computation.

In this chapter, we present a new algorithm, dubbed MetaOthello , for taxonomic classification of sequencing reads. Our algorithm builds upon taxon-specific k-mer signatures to support direct assignment to any level in the taxonomy. It employs a novel data structure, l-Othello, to support ultra-fast k-mer classification, achieving at least an order-of-magnitude improvement in speed over the state-of-the-art methods, Kraken and Clark, and three times faster than Kaiju. In the meantime, MetaOthello also substantially reduces the memory footprint, typically requiring only one third of the aforementioned methods. This modest memory requirement allows our algorithm to run on typical lab servers with 32 GB RAM, rendering it more accessible to biological researchers than those with memory requirements achievable only by supercomputers. Additionally, our algorithm is capable of conducting hierarchical top-down taxonomic classification and delivers performance competitive to, if not better than, other algorithms in both sensitivity and specificity as validated by benchmarking on a variety of datasets.

5.2 System Design of MetaOthello

5.2.1 k-mer Taxon Signatures

A k-mer is a length k subsequence of genomic sequences; for any sequence of length L, there exist a maximum of $L-k+1$ possible k-mers. Metagenomic reference material consists of one or more complete reference genomes belonging to an organism. Increasingly sophisticated sequencing techniques have permitted discovery of distinct reference genomes for a single species of organism, thereby capturing genomic variations that are often important to the functionality of the microbial species. The number of genomes (whether draft or complete) available as metagenomic reference material increases with each new discovery. If we consider each dataset as a collection of k-mers, a given taxon

can be described by the set of k-mers present in the reference sequences belonging to its taxonomic subtree. The problem of classifying a metagenomic read thus simplifies to the identification of the taxon that best matches the set of k-mers associated with the target read. When k is sufficiently large (*e.g.,* $k \geqslant 20$), the majority of k-mers are unique to the species carrying them. These species-specific k-mers may serve as signatures, directly implicating the appropriate taxonomic classification. However, a significant proportion of k-mers is present in multiple species, making them unique only to higher-ranking taxa. In this paper, we formalize the taxonomic specificity of a k-mer as the signature of a taxon: A k-mer is considered to be a **signature** of a taxon if (1) the k-mer does not appear in any genomic references belonging to ancestors or siblings of the target taxon, but only to sequences belonging to the taxon's subtree, and (2) the k-mer is not a signature of any lower-ranked taxon in the subtree. Equivalently, the taxon evincing a k-mer signature is the lowest common ancestor (LCA) of all species in the taxonomy whose reference genomes contain that k-mer.

In this way, as illustrated in Figure 5.1 and Figure 5.2, the set of all k-mers present in the genomic references of a taxonomy can be divided into disjoint collections, each of which contains the set of signature k-mers belonging to a single node in the taxonomy tree. Formally, let S be the set of all k-mers present in genomic references annotated by the taxonomy and let $T = \{1, 2, \cdots, |T|\}$ be the taxa (nodes) present in the taxonomy. Then S can be divided into $|T|$ disjoint sets, $S = \{S_0, S_1, \cdots, S_t, \cdots, S_{(|T|-1)}\}$, where for any node $t \in T$, S_t corresponds to the set of k-mer signatures belonging to taxon t. Thus, there exists a mapping, $g : S \to T$, such that $g(s) = t$ if the k-mer, $s \in S$, is a signature of the taxon, $t \in T$.

5.2.2 Taxonomic Classification of Sequencing Reads

As illustrated in Figure 5.2, given any sequencing read, our algorithm iterates over each k-mer from the beginning of the read and, for each k-mer, retrieves the taxon to which it is

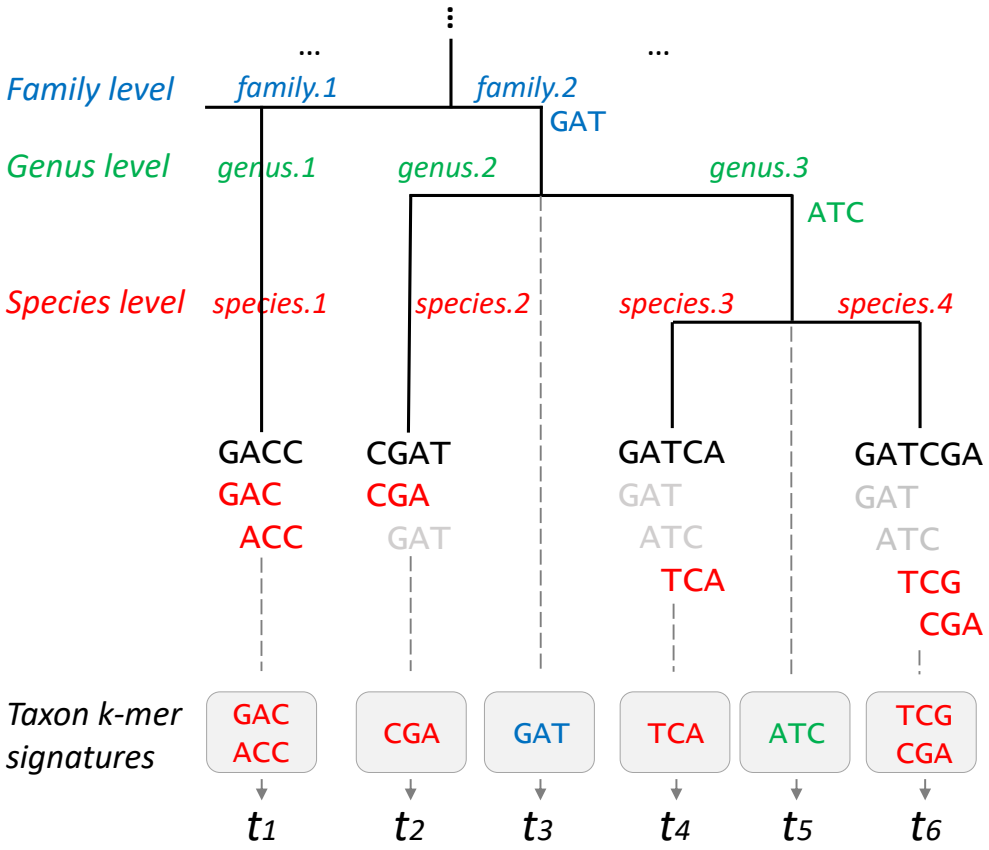


Figure 5.1: An example of MetaOthello taxonomy with reference sequences in the leaf nodes. The 3-mers that are signatures to each node are highlighted in read color.

specific using l-Othello. Taxonomic classification of the read is determined by assembling the taxa for all k-mers in the read. The classification is straightforward when all k-mers indicate the same taxon, but this is not often the case. Disparate taxa are considered to be consistent if they belong to the same path in the taxonomy, meaning that one assignment is the higher rank of the other. When these taxa belong to different branches, they represent conflicting information. The issue is further complicated by the possibility of false taxonomic information returned from querying alien k-mers, where the k-mer in the read does not appear in any of the reference sequences.

To tackle this challenge, we have designed a window-based classification approach. A window is defined as a sequence of consecutive k-mers that are assigned to the same taxon of a given level. The window-based approach guards against false-positive assignments due to alien k-mers. Assuming that the taxon ID returned by an alien k-mer is random, the

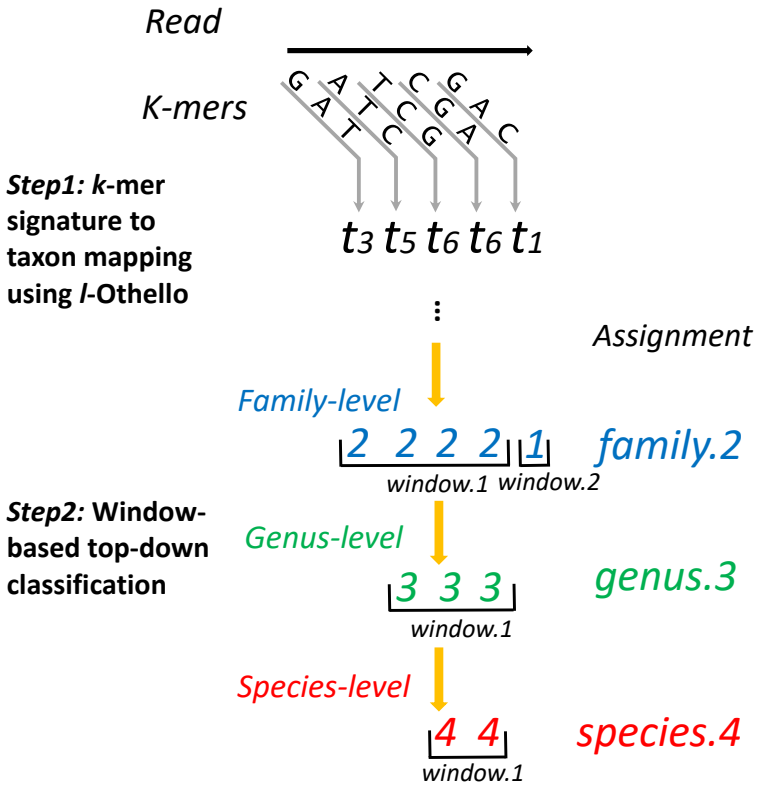


Figure 5.2: Step-by-step illustration of read classification on the taxonomy presented in Figure 5.1.

chance of having two consecutive alien k-mers return the same taxon ID is

$$\sum_{t=0}^{2^l-1} (p(t))^2 \sim 2^{-l}.$$

This value is very small, regardless of k. Additionally, each window corresponds to a maximum read subsequence that matches the reference sequences. Thus, the longer the window, the longer the subsequence match, and the less likely the match is random. In comparison, other algorithms such as Kraken and Clark count the total number of k-mer matches, regardless of their spatial distribution across the read.

If multiple taxon windows are available, MetaOthello scores each of them using the summed squares of window sizes as in the following formula; the taxon with the maximum score will be selected:

$$Score(t) = \sum (w_i^t)^2$$

where w_i^t denotes the number of k-mers in the i_{th} window classified to taxon t.

A k-mer signature belonging to a taxon is also specific to its higher-ranking taxa, so at higher taxonomic ranks, there exist more k-mers to distinguish a taxon from its siblings. Thus, longer k-mer windows and more accurate classifications are expected at higher taxonomic ranks. Under this assumption, a "top-down" strategy is adopted during read classification. Given a read sequence, MetaOthello starts the classification at the top rank and continues the classification down the ranks until there does not exist a sufficiently large k-mer window supporting the level. Based on the k-mer distribution in each taxon, MetaOthello establishes a threshold on minimum window-size when the classification on that taxon requires. Theorem 6 shows that the minimum window size threshold can be precomputed for each taxon prior to read classification. The minimum window size required for a taxon is determined by the probability of an alien k-mer query on l-Othello returning a taxon rooted in t and the acceptable false-positive rate. The larger the size of the taxon subtree, the higher the probability that a random alien k-mer may match to t and thus the longer the window required for reliable classification. Additionally, a larger window size will be required in order to lower the false-positive rate.

Theorem 6. *Given a user-defined false-positive rate λ and the total read number M, the minimum window-size threshold required for a taxon t can be computed as $\log_{p(t)} \frac{\lambda}{(1-\lambda)M}$, where p_t denotes the probability that an alien k-mer query on l-Othello returns a value in the taxon subtree with root t.*

Proof. We analyze the confidence of a K-mer window as follows. For a window of k-mers, let w be the length of the window. Suppose the query result for these K-mers are $\tau(s_1), \tau(s_2), \cdots, \tau(s_w)$. For a particular level of the taxonomy tree, suppose that these k-mer belongs to taxon t, then $\tau(s_1), \tau(s_2), \cdots, \tau(s_w) \in S_t$, where S_t is the set of the IDs of the nodes in the taxonomy subtree with the root t.

For consecutive w k-mers, let G_t be the event that this window of length w is from the taxon with ID t, without any sequence error. Let Q_t be the event that the query results of

these k-mers belongs to S_t, namely $\tau(s_1), \tau(s_2), \cdots, \tau(s_w) \in S_t$.

For a particular window of k-mers, let w be the length of the window, (i.e., there are $k+w-1$ bases in this window.

Let G_t be the event that this window is actually from taxon t. We assume there is no sequencing error, hence, when G_t the query results for these w k-mers satisfy :

$$\tau(s_1), \tau(s_2), \cdots, \tau(s_w) \in S_t$$

. We use notation Q_t to describe the event that $\tau(s_1), \tau(s_2), \cdots, \tau(s_w) \in S_t$.

Now the problem is that if we observe event Q_t, we may indicate two reasons exclusively. (1) Q_t happens as a result of G_t. (2) Note that for alien k-mers τ may return any integer, Q_t happens as a result of the query result of w alien k-mers. We use the probability $P(G_t|Q_t)$ to describe how confident we are, about that this window is from taxon t.

As described, when G_t happens, Q_t also happens. Hence $P(Q_t|G_t) = 1$.

We estimate the value of $P(G_t|Q_t)$ as follow.

$$P(G_t|Q_t) = \frac{P(Q_t|G_t)P(G_t)}{P(Q_t|G_t)P(G_t) + P(Q_t|\overline{G_t})P(\overline{G_t})} = \frac{P(G_t)}{P(G_t) + P(Q_t|\overline{G_t})P(\overline{G_t})} \tag{5.1}$$

Let q_t be the abundance of the window from taxon t. i.e., for a particular sample, randomly select one window of length w among all windows in all reads from this sample, the probability that this window is actually from taxon t. Hence $P(G_t) = q_t$.

The value $P(Q_t|\overline{G_t})$ is estimated as follow.

$\overline{G_t}$ means that this window is not from taxon t. $\overline{G_t}$ indicates either one of the following sub-events: (1) C_{other}: In this particular level of taxonomy tree, the window is from one other taxon t', which means the query results $\tau(s_1), \tau(s_2), \cdots, \tau(s_w) \in S_{t'}$ for a $t' \neq t$. Note that $S_{t'} \cap S_t = \emptyset$, this indicates $P(Q_t|C_{\text{other}}) = 0$. (2) C_{alien}: This window is an alien of the

taxonomy tree. Let $c_t = P(C_{\text{alien}}|\overline{G_t})$, then $0 < c_t < 1$.

$$P(Q_t|\overline{G_t}) = P(Q_t|C_{\text{other}})P(C_{\text{other}}|\overline{G_t}) + P(Q_t|C_{\text{alien}})P(C_{\text{alien}}|\overline{G_t}) = P(Q_t|C_{\text{alien}})c_t \quad (5.2)$$

As discussed in Section 2.2.3,

$$P(Q_t|C_{\text{alien}}) = q(t)^w \quad (5.3)$$

Combine Equation (5.1) (5.2) (5.3), we have

$$P(G_t|Q_t) = \frac{q_t}{q_t + p(t)^w c_t} \quad (5.4)$$

Note that, $q_t > 0$ and $0 < p(t) \ll 1$. Hence $P(G_t|Q_t) \to 1$ as $t \to \infty$. This is to say when w increases, $P(G_t|Q_t)$ also grows, and we can be more confident that when a query result shows that a window belongs to some taxon t, it reflects the fact that this window is actually from this taxon t. In other words, a longer window is more likely to come from this taxon than a shorter one.

Note that

$$\frac{q_t}{q_t + (p(t))^w c_t} > \frac{q_t}{q_t + (p(t))^w}$$

.

We use a threshold value λ, when $P(G_t|Q_t) > 1 - \lambda$, we accept, which is equivalent to:

$$w > \log_{p(t)} \frac{\lambda q_t}{(1-\lambda)} \sim \log_{p(t)} \lambda q_t$$

Here, the value of q_t can not be directly measured. However, for any actually detected taxon, we are sure that $q_t \geq \frac{1}{M}$, where M is the total number of reads in the dataset. Hence

we use the following threshold to decide the length of accepted windows.

$$w > \log_{p(t)} \frac{\lambda}{(1-\lambda)M} \sim \log_{p(t)} \frac{\lambda}{M}$$

Note that we can always use the l-Othello to compute the value of $p(t)$. Thus, given λ ($\lambda = 0.001$ by default), for each taxon, we can pre-compute the minimum size threshold for K-mer window. Only the K-mer windows which are not shorter than its associated minimum window size will be accepted for final assignment determination.

□

5.3 Comparison with the State-of-the-art Tools

We now assess the performance of MetaOthello in comparison to three of state-of-the-art tools: Kraken (version 0.10.5 beta), Clark (version 1.2.3), and Kaiju (version 1.4.4). Besides the newly published tool Kaiju, Kraken and Clark were chosen based on the recommendation of a recent benchmarking paper [105], which evaluated 14 tools using six datasets and subsequently declared Kraken and Clark the best performers over Genometa [106], GOTTCHA [107], LMAT [103], MEGAN [98][108], MG-RAST [109], the One Codex webserver, taxator-tk [110], MetaPhlAn [111], MetaPhyler [112], mOTU [113], and QIIME [114]. The comparison was benchmarked against three publicly available datasets: HiSeq, MiSeq, and SimBA5. The same datasets have been used multiple times to evaluate a number of metagenomic classification tools, including Kraken in previous studies [104]. All tools were executed using the same reference database (NCBI RefSeq as of October 1st, 2016), and all other parameters follow the default settings.

Runtime and Memory

Speed benchmarks were performed using the servers from Lipscomb High-Performance Computing at the University of Kentucky. The servers are equipped with Dell R820, Quad

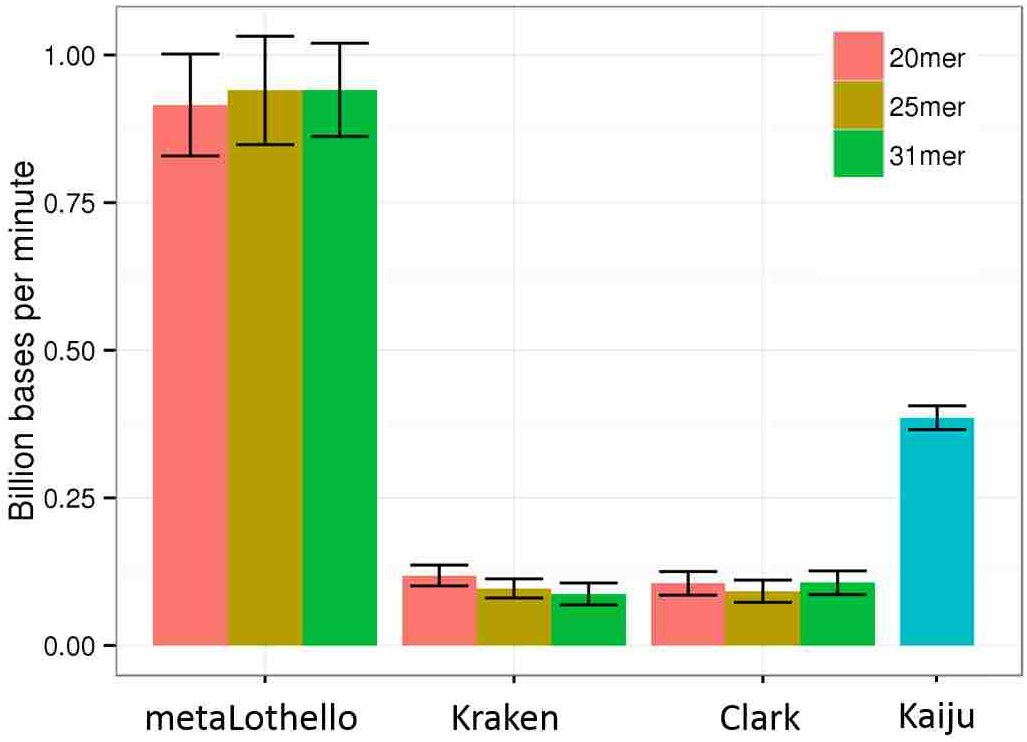


Figure 5.3: Billion bases processed per minute by each tool with three *k*-mer length settings using 8 threads.

Intel E5-4640 8-core (Sandy Bridge) @ 2.4 GHz and 512 GB/node of 1600 Mhz RAM. Each algorithm was executed using eight threads and *k*-mer lengths as specified previously; all other parameters follow the default settings. The speed for each tool is presented in Figure 5.3. In general, MetaOthello achieved the highest processing speed, clocking roughly 1 billion bases per minute. This figure represents an order-of-magnitude improvement over Kraken and Clark, the two most-rapid state-of-the-art tools within the category of alignment-free classifiers. Impressively, the high speed does not entail a compromise in the memory requirement. MetaOthello only consumes about one-third (peak memory 27 GB) the RAM required by Kraken and Clark (peak memory 73 GB).

The construction of the MetaOthello index from the NCBI RefSeq bacterial genome sequence database requires roughly 6 hours with peak memory usage up to 40 GB using 16 threads. In contrast, Kraken and Clark used 164 GB and 120 GB respectively for index

construction but both finished under 4 hours with 16 threads.

In summary, MetaOthello achieves a significant speedup with much smaller memory footprint in comparison with Kraken and Clark while delivering competitive or even superior performance in classification accuracy. While Kaiju is relatively scalable, it suffers from low sensitivity in classification.

Chapter 6. SeqOthello: Sequence Query Over Large Collections of RNA-seq Experiments

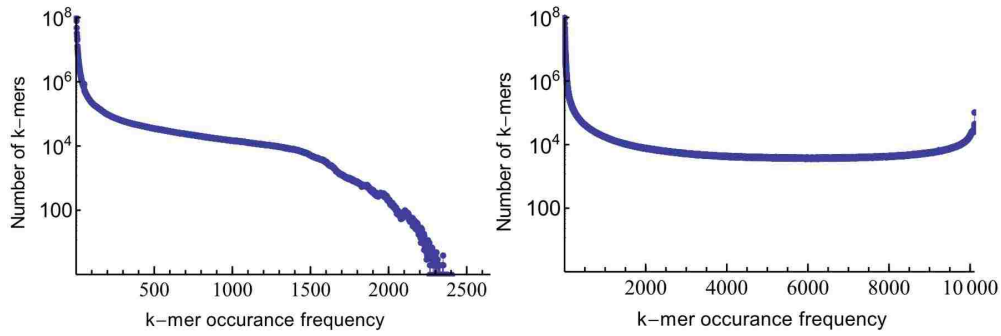
In this chapter, we present SeqOthello, an ultra-fast and memory-efficient indexing structure to support arbitrary sequence query against large collections of RNA-seq experiments. We first discuss the background of sequence query in Section 6.1. We present the system design of SeqOthello in Section 6.2. The algorithm for SeqOThello construction is presented in 6.3. We discuss the properties of SeqOthello during alien k-mer queries in 6.4. The evaluation results and clinical-significant finding of SeqOthello are presetned in Section 6.5.

6.1 Background

Advances in the study of functional genomics over the past decade have produced a vast resource of RNA-seq datasets. As of December 2017, over 12 Petabytes of RNA-seq data were deposited in the Sequence Read Archive (SRA)[19]. Sequencing consortiums such as The Cancer Genome Atlas (TCGA)[20] and the International Cancer Genomics Consortium (ICGC)[21] have sequenced tens of thousands of tumor transcriptomes from diverse cancer populations. Although these datasets have collectively redefined the landscape of cancer transcriptomes, additional clinically relevant features remain to be discovered. However, data reanalysis to identify these features requires extensive computational resources and bioinformatics support, making it exclusive to a few labs. The development of SeqOthello will enable labs with limited resources to learn from sequencing-level data by supporting fast and memory-efficient query over large-scale RNA-seq datasets.

To date, sequence search options are limited. Most sequencing databases support metadata searches[19, 21, 22, 31], which permit selection of experiments by tissue type, organism, experimental condition or sequencing protocol. From this refined list, experiments can be downloaded and analyzed individually [115]. Alternately, SRA-BLAST [116] can retrieve short reads aligned to a query sequence, but only for a limited number of nucleotides per query. Finally, the Bioinformatics community has lately established databases storing ready-to-analyze results in areas such as gene expression [22, 117, 118] and exon-exon junctions [119]. However, these databases are subject to frequent updates as Bioinformatics algorithms improve and reference genomes are refined, nor can they support the query of novel sequences that are absent from existing annotation or undetectable by current bioinformatics tools.

Recently, Sequence Bloom Tree (SBT) [120] and its descendants[121, 122] were developed to query RNA-seq experiments for expressed transcripts, pioneering the field of large-scale sequence search in RNA-seq. SBT is designed as an experiment filter that returns the subset of experiments containing at least θ percent of k-mers from the query sequence. Built upon bloom filters[123], SBT-based algorithms are generally memory efficient for small queries. Unfortunately, tuning the input parameter θ is time-consuming and produces inconsistent results for a single query, thereby hampering interpretability. Furthermore, extracting sequence-level information from the filtered experiments requires downloading and reanalyzing of raw sequencing datasets, and thus does not sidestep traditional RNA-seq processing. There is also growing interest in methods for indexing large collections of genomic sequencing reads from different individuals. Bloom filter trie (BFT) [124] was developed to store and compress a set of colored k-mers from a Pan-Genome of hundreds of samples. Additionally, the Burrows–Wheeler transform (BWT) and FM-index have been employed to build indexes on raw sequencing reads with applications in compressing 2705 whole genome sequencing samples from the 1000 Genomes Project [125, 126]. Though retaining full-text information, these data structures are often associated with high memory



a) 2,652 RNA-seq Experiments from SRA **b)** 10,113 TCGA Pan-Cancer RNA-seq Experiments.

Figure 6.1: The histograms of k-mer occurrence frequencies in two human RNA-Seq datasets. a) The k-mer occurrence histogram across 2652 RNA-seq experiments of human blood, breast and brain tissues from the SRA. b) The k-mer occurrence histogram across 10,113 TCGA Pan-Cancer RNA-seq experiments.

cost and slow query speed as the entire index must be loaded to memory prior to query.

6.2 System Design of SeqOthello

6.2.1 SeqOthello hierarchical structure

A sequencing experiment can be represented by a collection of k-mers, or length k subsequences of the original reads. k-mers are fundamental components of de Bruijn graphs and thus are essential for *de novo* assembly of the transcriptome[18–20] in individual experiments. A *database* of sequencing experiments can therefore be represented as a collection of *occurrence maps* of individual k-mers. The occurrence map of a k-mer is defined as its presence or absence across all experiments indexed in the database. The challenge is to efficiently store and query this information in scenarios with billions of k-mers across tens of thousands of experiments. We leverage novel algorithms in data compression and k-mer indexing to surmount this obstacle.

The prevalence of each k-mer varies dramatically, with plots of k-mer frequency often exhibiting a U- or L-shaped distribution. As shown in Figure 6.1, A k-mer's occurrence frequency is the number of samples containing the k-mer. The difference in the high frequency k-mers between the two datasets suggests less homogeneity in RNA-seq ex-

periments downloaded from SRA than these generated by TCGA. *k*-mers located at the extremes of the spectrum tend to originate from experiment-specific transcripts, or to descend from common transcripts that manifest in nearly all experiments. By contrast, *k*-mers near the center of the distribution may be tissue- or organism-specific. The prevalence of a *k*-mer directly determines the information content [127, 128], or the number of bits required to store its occurrence map. To this end, SeqOthello employs an information-content-aware data-compression scheme: an ensemble of compression techniques tailored to store the occurrence maps of *k*-mers from each region of the occurrence distribution without hampering query efficiency (Figure 6.2). SeqOthello relies on a novel, hierarchical indexing structure to facilitate fast retrieval of *k*-mer occurrence maps (Figure 6.2). The mappings between levels are supported by the Othello Hashing method. But an Othello constructed on billions of *k*-mers still demands too much memory to be practical for use with standard computers. The hierarchical structure employed by SeqOthello overcomes this challenge using a divide-and-conquer approach. Specifically, *k*-mer occurrence maps are split into buckets according to their encoded lengths, with the assignment of each *k*-mer to its bucket determined by the root Othello. Within each bucket, the mapping between a *k*-mer and the location of its occurrence map is again stored in an Othello. SeqOthello significantly increases the volume of indexed *k*-mers within limited memory space and is inherently parallelizable.

SeqOthello supports scalable *k*-mer searching in large-scale sequencing experiments. As shown in Figure 6.2, The bottom level of SeqOthello stores the occurrence maps of individual *k*-mers, encoded in three different formats and divided into disjoint buckets. The mapping between a *k*-mer and its occurrence map is achieved by a hierarchy of Othello structures in which the root Othello maps a *k*-mer to its bucket and the Othello in each bucket maps a *k*-mer to its occurrence map. b. An example illustrating SeqOthello's sequence-query process and output. A sequence query is decomposed into its constituent *k*-mers. The query result can be either a k-mer hit map, recording each *k*-mer's pres-

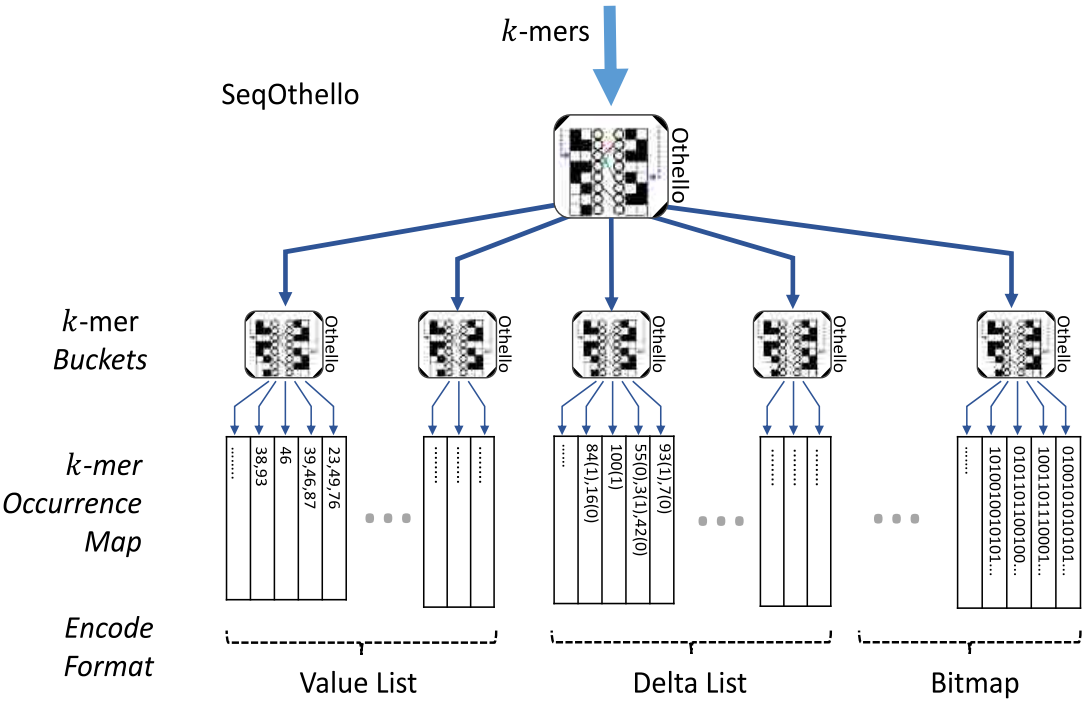


Figure 6.2: SeqOthello Indexing Structure

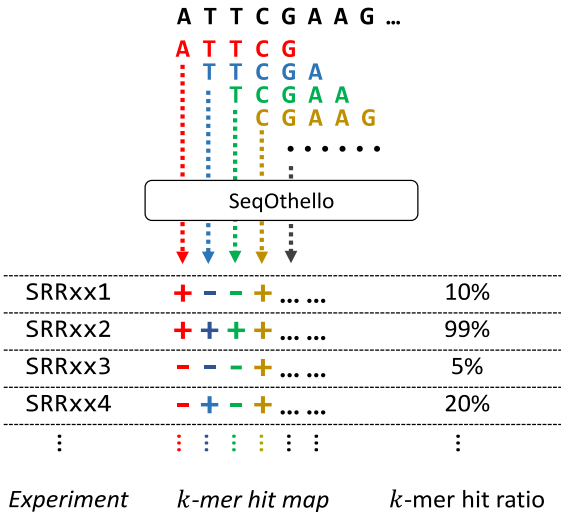


Figure 6.3: SeqOthello Query Procedure. $+/-$ indicate query hit or miss.

ence/absence along the query sequence, or k-mer hit ratios (i.e., the fraction of query k-mers present in each experiment)

Querying of a SeqOthello first requires decomposing the query sequence into its constituent k-mers. The root Othello node identifies the occurrence bucket for each k-mer, fol-

lowing which each bucket Othello node retrieves the desired occurrence map. Per k-mer, this process requires exactly two Othello queries and is thus executed in constant time. The full set of occurrence maps is then synthesized to generate a k-mer hit map of the query for each experiment, where a hit means a k-mer is present in an experiment. Each k-mer hit map can be summarized into the number of hits, or a hit ratio, the fraction of hits out of the total k-mers in the query (Figure 6.3).

6.2.2 Encoding of k-mer occurrence map

We define the occurrence map of a k-mer as a binary vector recording the k-mer's presence or absence in each experiment. Given m experiments, the occurrence map can be stored using m bits, where 1 represents presence and 0 represents absence in a certain experiment. To minimize the storage requirement of these vectors, we have developed a hybrid encoding method that leverages one of three different encoding strategies depending on the occurrence frequency of a k-mer. Each k-mer is stored using the method that yields the shortest code. These encoding methods are detailed below:

- **Value-list encoding.** This method is used to compress occurrence maps associated with rare k-mers. For an m-bit occurrence map with exactly t 1s (representing presence in t out of m samples), we enumerate the t indices of these positions as a list. Each index is represented by t integers, each $\lceil \log_2 m \rceil$ bits long. This list can also be viewed as a $t\lceil \log_2 m \rceil$-bit integer. Value-list encoding is used when $t\lceil \log_2 m \rceil \leq 64$.

- **Delta-list encoding.** This approach is employed for occurrence maps with a relatively larger number of 1s ($t\lceil \log_2 m \rceil > 64$). The m elements in the occurrence map can be considered as a succession of alternating subsequences of 0s and 1s. Thus the map can be represented by a list of $2w+1$ integers, $\langle x_1, y_1, x_2, y_2, \cdots, x_w, y_w, x_{w+1} \rangle$, representing the number of digits in each subsequence, where $x_1 \geq 0, x_{w+1} \geq 0$; $y_1, y_2, \cdots, y_w \geq 1$,$x_2, x_3, \cdots, x_w \geq 1$; and $x_1 + y_1 + x_2 + y_2 + \cdots + x_w + y_w + x_{w+1} =$

Table 6.1: Hexadecimal encoding for integer values in delta-list encoding

Integer value z	Encoded binary representation	Hexadecimal value	Encoded length in bits
$0 \leq z < 8$	$(1xxx)_2$	$0x8 \mid z$	4
$8 \leq z < 64$	$(01xxxxxx)_2$	$0x40 \mid z$	8
$64 \leq z < 512$	$(001xxxxxxxx)_2$	$0x200 \mid z$	12
$512 \leq z < 4096$	$(0001xxxxxxxxxxxx)_2$	$0x1000 \mid z$	16
$4096 \leq z$	$(0000xxxxxxxxx\ldots)_2$	$0x0000 \mid z$	32

m. The occurrence map can be reconstituted by enumerating x_1 0s, followed by y_1 1s, x_2 0s, y_2 1s, etc. For example, consider an occurrence map of $m = 20$ elements, 1110011...10, with 1s at indices 1,2,3,6,8,9,...,19. The corresponding delta-list representation is $\langle x_1 = 0, y_1 = 3, x_2 = 2, y_2 = 14, x_3 = 1 \rangle$.

The $2w+1$ integers from this first step are further encoded as positive integers. Multiple procedures exist for the second encoding step, the choice of which depends on the relative importance of minimizing encoding/decoding overhead *versus* maximizing the compression rate. To balance the time and memory complexity of encoding, as well as the storage overhead, we choose to encode the delta list as a hexadecimal stream. Each integer is converted to a hexadecimal value using the method described in Table 1. We then concatenate the hexadecimal values into a single hexadecimal datum. For the delta list shown in the example, $\langle 0, 3, 2, 14, 1 \rangle$, the corresponding hexadecimal format is 0x8, 0xB, 0xA, 0x4E, 0x9. After concatenation, the final result is 0x8BA4E9.

- **Bitmap encoding.** Each occurrence bitmap is an m-bit value, with each bit coding the presence or absence information for one of the m samples. As this method requires more memory than other options, it is used only when other options do not work.

6.3 SeqOthello Construction Procedure

6.3.1 The Construction Algorithm

Construction of a SeqOthello data structure requires as input a list of k-mer files, each containing the set of k-mers extracted from reads associated with a distinct RNA-seq experiment. Currently the k-mer file is generated by applying Jellyfish to fastq files.

- **Step 1: Assembling the occurrence map of each k-mer in the collection of experiments to be indexed.** The goal of step 1 is to determine each k-mer's presence/absence information across all experiments. This task requires the integration of k-mers from all k-mer files, but simultaneous file access is time-consuming and not allowed by many operating systems. Instead, we employ a strategy similar to merge sort. We first obtain k-mer occurrence maps for small groups of experiments, where each group contains approximately 50 samples. These intermediate occurrence maps are encoded as delta lists, which significantly reduces file sizes. The groups are then merged to obtain the k-mer occurrences across all experiments. After SeqOthello is constructed, the group files generated at this step are no longer needed. However, as these files are orders of magnitude smaller than the original k-mer files, they can be stored to support update of the SeqOthello structure.

- **Step 2: Assignment of k-mer occurrence maps to buckets.** We next divide the entire set of k-mers into disjoint buckets based on their occurrence maps using the following principles: (1) Occurrence maps within the same bucket should be generated by the same encoding approach; (2) the lengths of encoded occurrence maps within the same bucket should have limited variation; and (3) the total size of the encoded occurrence maps within each bucket should not exceed a specified threshold (by default, 128 MB).

  Given a maximum bucket size, we define the range of encoding lengths for each

bucket prior to allocating k-mers. Note that the distribution of k-mer encoding lengths is unknown prior to construction. To avoid multiple iterations over all k-mers during bucket assignment, we designed a sampling-based approach to estimate the range of encoding lengths. The goal is to set an open upper bound n_{t+1} and closed lower bound n_t so that k-mers with encoding lengths in the range $[n_t, n_{t+1})$ are assigned to each bucket t. We select 10 million k-mers, which is approximately 0.1% of the k-mers present over all experiments, and let L_i be the estimated number of k-mers with encoding length equal to i. Starting from $t = 1$ and $n_1 = 1$, we greedily select the maximum index n_{t+1} so that $n_t L_{n_t} + (1+n_t)L_{1+n_t} + \cdots + (n_{t+1}-1)L_{n_{t+1}-1} \leq 128M$. Once the number of buckets and their ranges of encoding lengths are determined, the construction algorithm will iterate over each k-mer, assigning it to the appropriate bucket in accordance with the encoding length of its occurrence map. The encoded occurrence maps are further compressed by gzip when the final structure is stored as a file.

- **Step 3: Establish k-mer mapping using Othello.** During step 2, SeqOthello maintains the list of k-mers and their corresponding encoded occurrence maps in each bucket. Once the k-mer assignment is completed in the bucket, an Othello will be established to record the mapping between k-mers and the locations of their occurrence maps. Once the buckets are finalized, a root Othello is constructed to record the mapping between the entire set of k-mers and their bucket IDs.

  SeqOthello also maintains an .xml file to store metadata associated with the data structure, which includes basic information about the experiments and information necessary for the query algorithm to interpret the data file.

6.3.2 Optimization for k-mers that appear in only one experiment

The prevalence of individual k-mers varies dramatically, with plots often exhibiting a U- or L-shaped distribution (Figure 6.1). Note that the number of k-mers present in only one

experiment is relatively large compared to k-mers with higher frequencies. We apply the following approach to improve the efficiency and accuracy of SeqOthello.

Instead of storing all k-mers with single occurrence in a level-2 bucket, we encode them directly in the root Othello. Let E be the set of experiments indexed by SeqOthello, identified by integers $\{1,2,\cdots,|E|\}$. Let B be the set of buckets identified by integers $\{|E|+1,|E|+2,\cdots,|E|+|B|\}$. The root Othello records the mapping between k-mer set S and $E \cup B$. For any k-mer s, if the query result on the first level $\tau(s) \in \{1,2,\cdots,|E|\}$, SeqOthello will report that s is present in the experiment with index $\tau(s)$; if $\tau(s) = |E| + b$ for some integer $b \in \{1,2,\cdots,|B|\}$, then $\tau(s) \in B$ and the query process will continue into the bucket with index b on the bottom layer of SeqOthello.

6.3.3 Insertion of new experiments into SeqOthello

If the group files generated at Step 1 have been retained, the insertion of new experiments to SeqOthello is quite fast, especially for batch update. The process involves merging newly inserted experiments with the existing group files, and then repeating Steps 2 and 3 of the above construction algorithm. The entire update requires only a few hours to complete.

6.4 False-positive k-mer Query on SeqOthello

SeqOthello maintains a mapping from a large set of k-mers to their occurrence maps. However, due to the nature of Othello being a minimal perfect hashing classifier, querying of an alien k-mer (*i.e.*, k-mer that does not exist in any of the samples) with SeqOthello may afford a false report of its presence in one or more RNA-seq experiments. Here, we analyze the likelihood of such a false report.

6.4.1 Notations

In reference to SeqOthello, we use the notation ${}^{\text{Root}}\boldsymbol{O}(S,V)$ to denote the root-level Othello. ${}^{\text{Root}}\boldsymbol{O}(S,V)$ records the mapping between a k-mer in S and its assignment either to a single

Table 6.2: A summary of notations used in Section 6.4.1

${}^{\text{Root}}\boldsymbol{O}(S,V)$	Othello at the root of SeqOthello
${}^{b}\boldsymbol{O}(S_b,V_b)$	Othello of the bucket b
E	Set of RNA-seq experiments
B	Set of buckets
W_t	t^{th} occurrence map in a bucket b
${}^{\text{SeqOthello}}P_{\text{Alien}}$	Probability of an alien k-mer being recognized as Alien by SeqOthello
${}^{\text{SeqOthello}}P(e)$	Probability of an alien query returning experiment e
${}^{\text{root}}p_x$	Probability that query of an alien k-mer on the root Othello $\tau(s')$ returns x
${}^{b}p_x$	Probability that query of an alien k-mer on the Othello in bucket b returns ${}^{b}\tau(s')$ value x

experiment or to a second-level bucket in $V = E \cup B$.

For any bucket $b \in B$, we use the notation ${}^{b}\boldsymbol{O}(S_b,V_b)$ to denote the associated Othello, where ${}^{b}\boldsymbol{O}(S_b,V_b)$ stores the mapping between a k-mer in S_b and its occurrence map index in V_b. Thus S_b is the set of k-mers that are assigned to bucket b and $V_b = \{1,2,\cdots,v_b\}$ is the list of indices for encoded occurrence maps in bucket b.

We list the primary notation used in the following analysis in Table 2.

6.4.2 Probability of alien k-mer recognition and false positive presence

Let s$'$ be an alien k-mer, and $\tau(s')$ be the result returned when querying s' on the root Othello. Then $\tau(s')$ falls into one of the following three categories:

A. $\tau(s') \notin V$, where $V = E \cup B$. This k-mer will be identified as alien, and SeqOthello will report its absence from the database. The probability of this result is ${}^{\text{root}}P_{\text{Alien}}$, which can be calculated according to Theorem 1.

B. $\tau(s') \in E$. Such a k-mer will be reported falsely as existing in the experiment identified by $\tau(s')$. For any experiment $e \in E$, the probability of returning e as the result of querying an alien k-mer has a probability ${}^{\text{root}}p_e$, which can be calculated based on Lemma 1.

C. $\tau(s') \in B$. In this case, the query process would continue into the bucket b identified by $\tau(s')$. This circumstance occurs with probability ${}^{\text{root}}p_{|E|+b}$. Inside the bucket b, the query ${}^{b}\tau(s')$ will result in one of two scenarios:

1. ${}^{b}\tau(s') \notin V_b$. In this case, s' is identified as alien in bucket b with probability ${}^{b}P_{\text{Alien}}$, which is P_{Alien} for the Othello ${}^{b}\boldsymbol{O}(S_b, V_b)$.
2. ${}^{b}\tau(s') \in V_b$. Here $s^{'}$ is mapped falsely to a location storing the occurrence map of a different k-mer. A calculation follows for the probability of this outcome.

Assume there are v_b encoded occurrence maps stored in bucket b, namely $W_1, W_2, \cdots, W_{v_b}$. We use the notation $W_{t,e} \in \{0,1\}$ to denote the presence/absence information for experiment e stored in the t-th occurrence map. Here, $W_{t,e} = 1$ indicates that the k-mer associated with occurrence map W_t is marked as 'present' in experiment e; $W_{t,e} = 0$ indicates it is marked as 'not present' in experiment e.

Note that a query on bucket b returns the occurrence map with index ${}^{b}\tau(s^{'})$, namely $W_{{}^{b}\tau(s')}$. For any experiment e, $1 \leq e \leq |E|$, if $W_{{}^{b}\tau(s'),\mathbf{e}} = 1$, then the query result would indicate falsely that s' presents in experiment e. We use the notation ${}^{b}P(e)$ to denote the probability of the query on bucket b yielding $W_{{}^{b}\tau(s'),\mathbf{e}} = 1$. ${}^{b}P(e)$ is equal the probability of ${}^{b}\tau(s')$ returning any index x such that the x-th occurrence map W_x satisfies $W_{x,e} = 1$:

$$ {}^{b}P(e) = \Pr\left[W_{{}^{b}\tau(s'),e} = 1\right] = \sum_{x \in V_b} \Pr\left[{}^{b}\tau(s') = x \wedge W_{x,e} = 1\right] $$

Noting that $W_{x,e} \in \{0,1\}$,

$$ {}^{b}P(e) = \sum_{x \in V_b} {}^{b}p_x W_{x,e} $$

Computing ${}^{b}p_x$ for all $x \in V_b$ using Lemma 1 requires $O\left((2^{l_b})^2 |V_b|\right)$ computation, which becomes infeasible when l_b is large. Hence, we use an alternative approach to estimate the ${}^{b}p_x$ values when $l \geq 12$. Lemma 2 indicates that the value of ${}^{b}p_0$ is significantly

larger than ${}^{b}p_x$values for $x \neq 0$. We also observe that the values for ${}^{b}p_x$ are similar for any $x \neq 0$ and $x < 2^{l_b}$ in the same bucket b. We therefore use the average value of ${}^{b}p_x$ over $x \neq 0$, denoted by ${}^{b}p_{x\neq 0}$, to replace individual ${}^{b}p_x$ values:

$$ {}^{b}p_{x\neq 0} = \frac{1}{2^{l_b}-1}(1-{}^{b}p_0) $$

Hence,

$$ {}^{b}P(e) = \sum_{x\in V_b} {}^{b}p_x W_{x,e} \rightarrow \sum_{x\in V_b} {}^{b}p_{x\neq 0} W_{x,e} = {}^{b}p_{x\neq 0} \sum_{x\in V_b} W_{x,e} = \frac{\left(1-{}^{b}p_0\right)}{2^{l_b}-1} \sum_{x\in V_b} W_{x,e} $$

Here, $\sum_{x\in V_b} W_{x,e}$is the number of encoded occurrence maps in bucket b in which the associated k-mer is marked to be present in experiment e.

For an alien k-mer s', the query on SeqOthello may return a false presence in experiment e if $\tau(s')$ falls in category B, a circumstance which occurs with probability ${}^{\text{root}}p_e$. Otherwise, if $\tau(s')$ satisfies circumstance C.2, the query yields an occurrence map in which experiment e is marked as positive with probability ${}^{b}P(e)$. Hence, the probability of an alien k-mer query on the two-level SeqOthello yielding a false-positive presence in experiment e is:

$$ {}^{\text{SeqOthello}}P(e) = {}^{\text{root}}p_e + \sum_{b=1}^{|B|} {}^{\text{root}}p_{|E|+b} \cdot {}^{b}P(e) $$

On the other hand, an alien k-mer has a very good likelihood of being recognized as alien if $\tau(s')$ satisfies circumstance A, or falls in circumstance C and is subsequently identified under C.1. Taken together, the overall probability of SeqOthello identifying the k-mer as alien is:

$$ {}^{\text{SeqOthello}}P_{\text{Alien}} = {}^{\text{root}}P_{\text{Alien}} + \sum_{b=1}^{|B|} {}^{\text{root}}p_{|E|+b} \cdot {}^{b}P_{\text{Alien}} $$

Table 6.3: Estimated probability values computed on SeqOthello constructed for Human and TCGA datasets

	SRA	**TCGA**
$\|E\|$: number of experiments	2,652	10,113
$\|B\|$: number of buckets	105	127
${}^{\text{SeqOthello}}P_{\text{Alien}}$	0.532440	0.551722
${}^{\text{SeqOthello}}P(e)$, average over all experiments	0.000840	0.000606
standard deviation of ${}^{\text{SeqOthello}}P(e)$, across all experiments	0.000684	0.000173

We present a numerical estimation of various probabilities based on the distribution of k-mer occurrences as well as the SeqOthello structures constructed for the two datasets used in this paper. The results are given in Table 6.3.

6.4.3 Error rate of a SeqOthello sequence query

SeqOthello executes sequence query by making individual k-mer queries extracted from the sequence. The probability of returning false-positive k-mer hits is low and can be computed as ${}^{\text{SeqOthello}}P(e)$. Let $X(e)$ be the number of false positives for experiment e returned over w alien k-mer queries. Then, $X(e)$ follows the binomial distribution $Binomial\left(w,{}^{\text{SeqOthello}}P(e)\right)$. Note that the query result for transcript query is reported as the fraction of present k-mers for each sample, and $X(e)$ false positive k-mers will result in an error rate of $\frac{X(e)}{w}$. Note that the $\frac{X(e)}{w}$ is usually 0. The probability of $\frac{X(e)}{w}$ being large enough to affect the query result is very low, only occurring when multiple k-mer queries return the same false-positive experiments. For example, for $w = 50$ and $P(e) = 0.0084$, the probability of $X(e) > 2$ is 1.15×10^{-5}. Thus, SeqOthello returns the query result with error rate $\delta = \frac{X(e)}{w} > \frac{2}{50} = 4\%$ with probability 1.15×10^{-5}, which is much lower than the probability of a single error.

Table 6.4: Performance comparison on construction.

Tool	SeqOthello	SBT	SBT-AS	SSBT
k-mer preparation (days)	3.4	4.1	4.3	4.8
Index building (hours)	1.9	39.5	10.2	46.6
Peak memory (GB)	14.1	23.4	39.1	5.6
Intermediate disk space (TB)	0.9	1.4	3.7	1.9
Final index size (GB)	20.8	185.5	168.5	30.8

6.5 Evaluation

6.5.1 On Query performance

We compare SeqOthello to each of three state-of-the-art methods for querying large-scale RNA-seq datasets: SBT [120], SSBT [121], and SBT-AS [122]. The evaluation was benchmarked on 2,652 RNA-seq experiments of human blood, breast, and brain tissues from the SRA. The results are shown in Table 6.4. The k-mer preparation step converts each individual sequencing experiment to the binary format using 16 threads. We use Jellyfish [129] to convert raw sequence data into *k*-mer files. In order to alleviate noise from sequencing errors, k-mers having a frequency lower than a specific threshold were removed from the experiment (Table 6.5). The thresholding criteria are only applied to the 2,652 human RNA-seq experiments from SRA. Only k-mers with frequency count no less than the threshold are retained for subsequent indexing.

The index step follows the k-mer prep step. Unless mentioned otherwise, the comparisons were tested using a single thread. SeqOthello reduces the index construction time by 81% comparing to SBT-AS and the final index size by 32% comparing to the smallest SSBT index. As shwon in Table 6.6, taking these files as input, SeqOthello requires 1.93 hours and a maximum of 14.1 GB memory to construct the index, 10 times faster than SBT and SSBT. At 20.8 GB, the SeqOthello index achieves a 700:1 compression ratio relative to the original database.

The time cost of SeqOthello is shown in Figure 6.4. The memory cost of SeqOthello

Table 6.5: k-mer count threshold used to obtain k-mers from Jellyfish as a function of the fasta.gz file size of each experiment

fasta.gz size	Threashold
$<$ 300 MB	1
Between 300 MB and 500 MB	3
Between 500 MB and 1GB	10
Between 1GB and 3GB	20
$>$ 3GB	50

Table 6.6: Query response time of SeqOthello, SBT, SSBT, and SBT-AS

	Memory (GB)	Time (min)
SeqOthello (1 thread)	15.2	35.7
SeqOthello (4 thread)	19.4	13.4
SBT	22.4	4160.6
SBT-AS	61.0	575.5
SSBT	2.8	6964.6

is shown in Figure 6.5. SeqOthello queries 198,093 transcripts from Gencode Release 25 [130] for k-mer hits in all 2,652 experiments in 35.7 minutes using 15.2 GB memory. With four threads, the running time drops to 13.4 minutes. SBT-based queries only return the set of experiments whose k-mer hit ratio is greater than a user-defined threshold, denoted by θ. Even with a very high k-mer hit ratio ($\theta = 0.9$), SBT-AS and SBT require 575 and 4,160 minutes to complete, respectively with higher memory cost than SeqOthello. While SSBT is extremely memory frugal, it is at the expense of much slower speed, two orders of magnitude slower than SeqOthello.

The detailed comparison is shown in Table 6.7. The significance of experiments extracted by SBT using a single threshold θ is difficult to assess. To avoid generating misleading conclusions, multiple queries with different θ may be attempted to determine an approximate distribution, affording an overall query time several times larger than we report. Querying a small batch of 1,000 transcripts with settings of $\theta = 0.7$, $\theta = 0.8$, and $\theta = 0.9$ required 40 minutes to execute with SBT-AS, 190 minutes with SBT, and 241 minutes with SSBT. In contrast, SeqOthello requires only 4.6 minutes to query the same set of

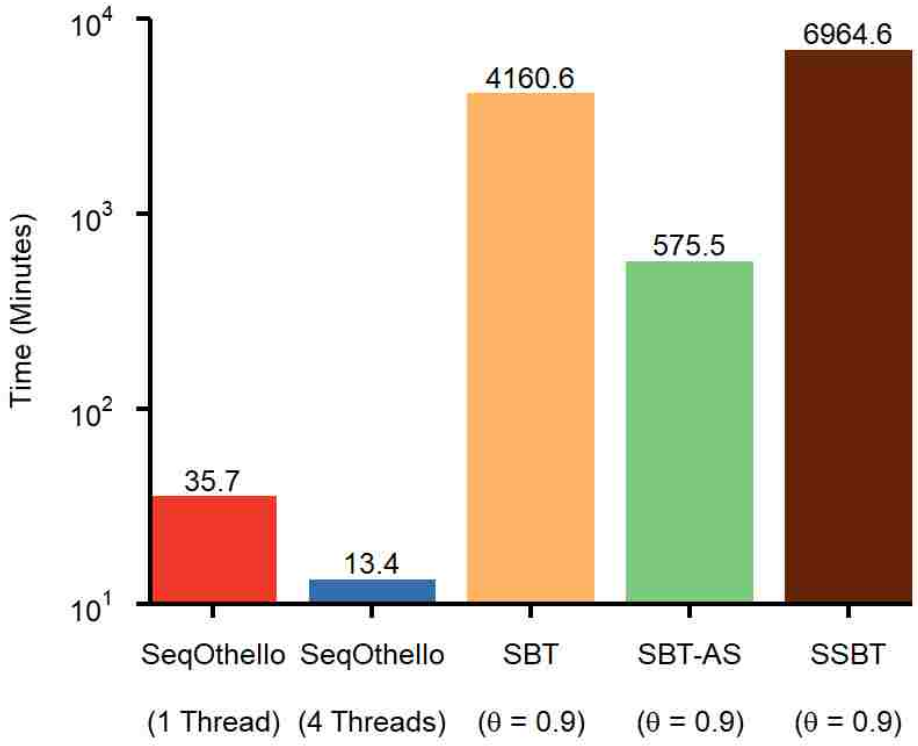


Figure 6.4: Query response time of SeqOthello, SBT, SBT-AS, and SSBT

transcripts, and generates exact hit ratios for each transcript in each indexed experiment.

SeqOthello also accommodates online features for small-batch queries. Online queries preload the entire index into memory prior to querying, and can be executed in approximately 0.09 seconds per transcript. Our method's advantageous speed permits it to support on-demand and instant queries from multiple users in a client-server setting. Other methods do not have online options at present.

6.5.2 On Query Accuracy

SeqOthello always returns the correct occurrence map when querying k-mers from the set the SeqOthello is built upon. This includes the set of k-mers that are present in at least one experiment sample. However, for queries involving alien k-mers that are not present in any of the experiments indexed in the database, SeqOthello may return false positive occurrences. To assess the accuracy in general k-mer search, we queried 120,044,842 k-

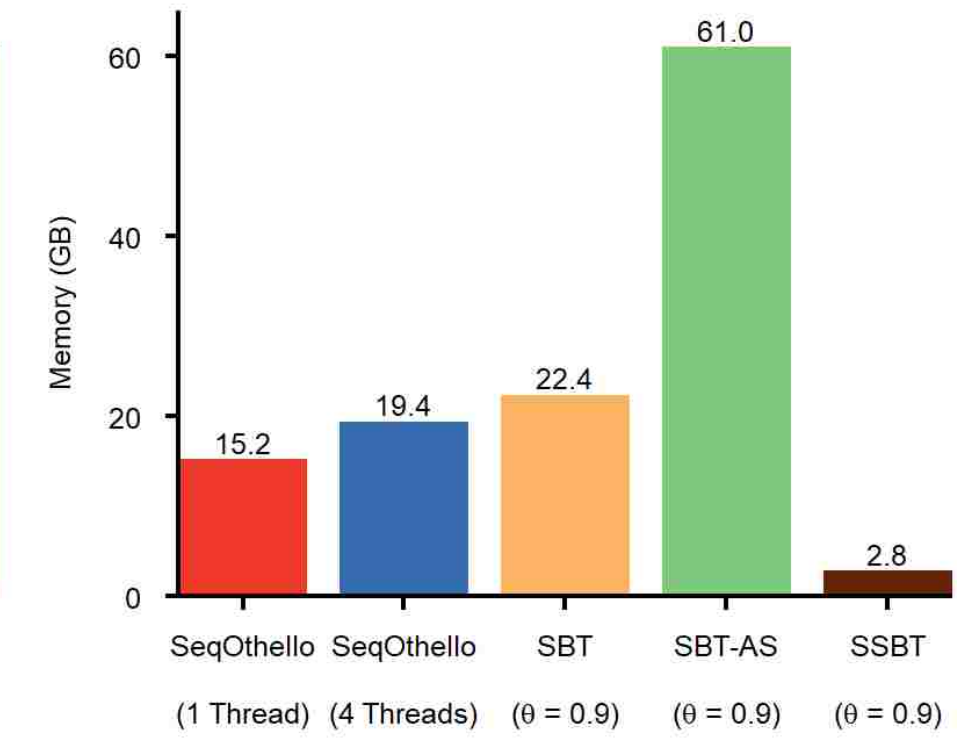


Figure 6.5: Peak memory usage of SeqOthello, SBT, SBT-AS, and SSBT

mers present in human transcriptome Gencode Release 25 against the SeqOthello index constructed for the aforementioned 2,652 experiments. We randomly selected 150 experiments and calculated the false-positive rate of k-mer queries in each experiment. The false positive rate is defined as the fraction of k-mers absent from the raw k-mer file that SeqOthello classifies as present with all queried k-mers. SeqOthello recovers all k-mers that are truly present in the experiment, with guaranteed 100% recall rate. For k-mers that are not present in any of the indexed experiments, SeqOthello yields an extremely low rate of false positives: across 150 randomly chosen experiments, the average false-positive rate was only 0.015% with standard deviation of 0.071%.

To further evaluate the effect of false positives on transcript queries, we mapped the raw k -mers of each experiment to transcript sequences, calculating the true k-mer hit ratio for each transcript. We then compared the k-mer hit ratios generated by SeqOthello to the ground truth.

Table 6.7: Performance comparison on small batch query.

	max filters	Memory (GB)	Time (min)	
SeqOthello-online	-	-	27.3	1.5
SeqOthello	-	-	6.1	4.6
SBT	.7	1	0.5	78.4
	.8	1	0.5	64.7
	.9	1	0.5	46.9
	Total	1	0.5	190.0
	.9	10,000	175.1	48.9
SBT-AS	.7	1	0.8	14.5
	.8	1	0.8	13.0
	.9	1	0.8	12.5
	Total	1	0.8	40.0
	.9	10,000	131.4	11.6
SSBT	0.7	1	0.2	100.2
	0.8	1	0.2	82.7
	0.9	1	0.2	58.6
	Total	1	0.2	241.5
	0.9	10,000	28.1	55.6

As shown in Figure 6.6, the error (δ) of a transcript query over an experiment is calculated as the difference between the transcripts k-mer hit ratio returned by SeqOthello and the k-mer hit ratio obtained by mapping raw k-mers using the same RNA-seq experiments to the transcript sequences. Each bar shows the percentage of transcripts with δ falls in a particular range. The error bar shows the standard deviation of such percentage measured on 150 experiments. . Roughly 89.7% of transcripts afforded k-mer hit ratios equal to the true value, with an additional 9.3% exhibiting an error rate up to 0.003. These results demonstrate that SeqOthello achieves near-exact query of k-mers and k-mer hit ratios. Additionally, as consecutive k –mers in a sequence are highly redundant, even a single base mismatch to the query sequence will be evidenced by the absence of multiple (i.e. k) k -mers, rendering an extremely low likelihood of false positive match due to alien k-mers

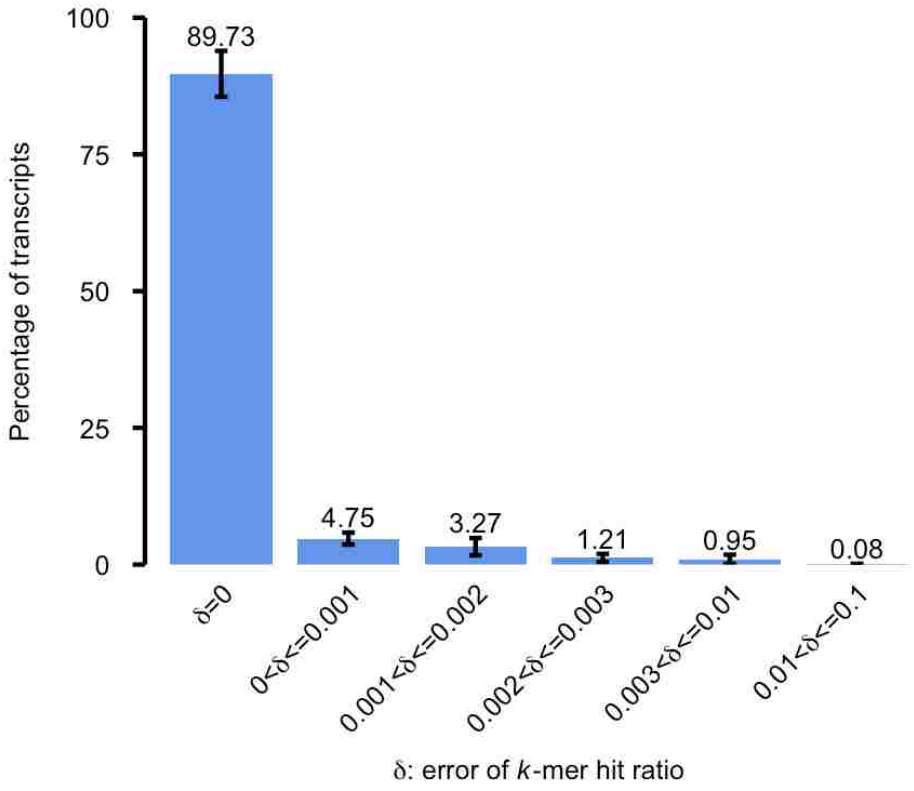


Figure 6.6: The distribution of error rate in k-mer hit ratios returned by SeqOthello

in sequence query (Methods). Although *k*-mer information is implicitly stored in bloom filters employed in SBT-based algorithms, efficient implementation of *k*-mer retrieval by these algorithms is not yet available.

6.5.3 Anlaytical result on query over TCGA Pan-Cancer RNA-Seq Experiments

The Cancer Genome Atlas (TCGA)[20] contains transcriptome profiles of 10,113 tumor samples obtained from 9,215 cancer patients. The database allows researchers to detect and characterize novel transcriptomic alterations across 29 different cancer types in the GDC Legacy Archive. We have constructed a SeqOthello index, storing the occurrences of 1.47 billion 21-mers across all tumor samples. The preparation of *k*-mers averages 4 minutes per sample while the construction of SeqOthello on all samples took less than 9 hours. The index occupies only 76.6 GB of space, thus is portable for querying at different locations.

We use the SeqOthello index to conduct a survey of all gene-fusion events curated by TCGA Fusion Gene Database as of December 2017[131]. The database documented of 11,658 unique tier-1 fusion events from TCGA detected by PRADA[132]. This represents 10,994 gene fusion pairs as multiple junctions might exist for one fusion pair.

We use the fusion junction sequence for fusion query on SeqOthello. Each fusion junction sequence consists of 20 bases from donor exon in one gene and 20 bases from acceptor exon in the other gene. Each 21-mer within this 40-base sequence spans the fusion junction. The query of a fusion sequence using SeqOthello may return a maximum of 20 k-mer hits for each RNA-seq Experiments indexed by SeqOthello.

A SeqOthello query of a fusion sequence returns the number of k-mer hits in each sample. A simple method to determine the fusion occurrence in each sample can be done in SBT-like approach, where a minimum fraction of k-mer hits, θ, is required to call the presence. However, this technique yields lackluster sensitivity and specificity. Lowering θ permits fusion detection with fewer spanning reads, but may increase false-positive calls if the fusion junction sequence contains repetitive k-mers that are abundant in many samples. Instead of using a fixed threshold for all fusion calls, we develop a noise-aware approach. This approach first evaluates the background noise of the query result due to repetitive k-mers that are abundant in large fraction of samples, which can be detected leveraging the distribution of k-mer hits across TCGA tumor samples queried through SeqOthello.

Under this method, we detect 92.7% of tier-1 fusion occurrences in TCGA Fusion Gene Database[27] with at least 10 spanning reads reported by PRADA. Additionally, we identify 270 novel occurrences of fusion events across 17 tumor subtypes that are not identified by PRADA. We selected two fusion pairs with occurrences most inconsistent with current curation for further validations: FGFR3-TACC3 in GBM samples (5 novel, 3 undetected) and ESR1-C6orf97 in BRCA samples (2 novel, 5 undetected). We were able to confirm all 7 novel fusion occurrences by the identification of at least 10 fusion spanning reads supporting each. For all undetected fusions, insufficient spanning reads were confirmed,

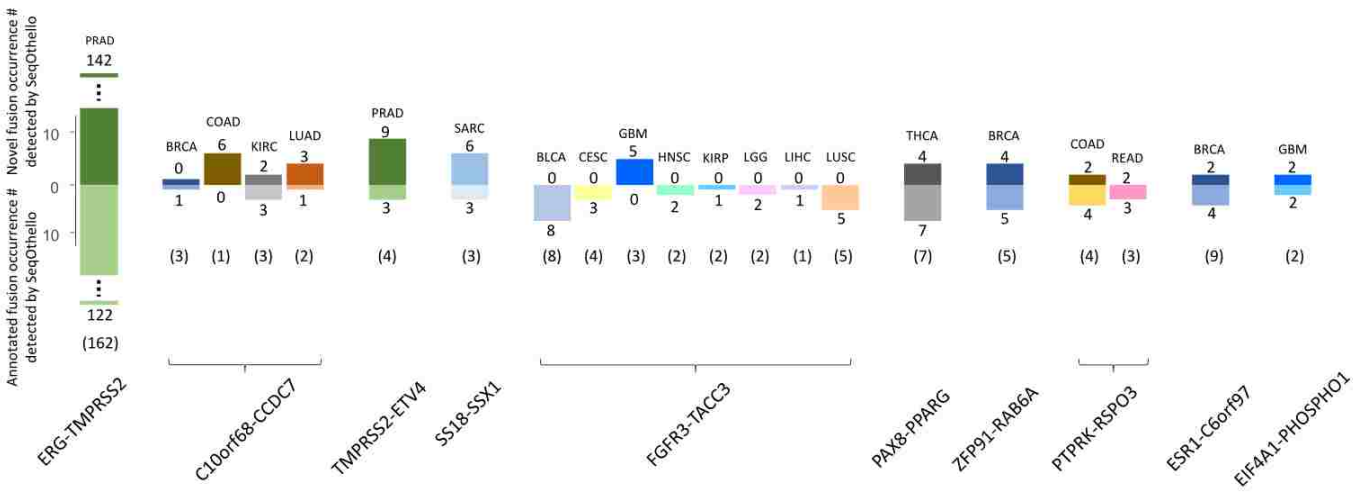


Figure 6.7: Ten fusion gene pairs with the highest novel occurrences identified by SeqOthello

which are consistent with low read support recorded in the database.

Figure 6.7 shows ten fusion gene pairs with the highest novel occurrences identified by SeqOthello. We used SeqOthello to conduct a survey of all fusions curated in TCGA gene-fusion database against 10,113 TCGA Pan Cancer RNA-seq samples. Each column in the bar plot represents fusion occurrence in a cancer type. The number in parenthesis below each bar indicates the number of documented occurrences reported in the current database. Interestingly, all novel occurrences agree with the original fusion cancer-type classifications, rendering the chance of random occurrence negligible. This result corroborates their cancer specificity and supports the high precision of SeqOthello's query results. One example of this consistency is TMPRSS2-ERG, a clinical marker for prostate cancer. SeqOthello extracted 122 pre-identified occurrences of TMPRSS2-ERG and 142 novel occurrences, all from prostate cancer samples.

6.6 Discussion

SeqOthello is a novel algorithm capable of indexing large-scale RNA-seq experiments that supports online sequence query. We constructed a SeqOthello index on the TCGA Pan-Cancer RNA-seq datasets, the latter totaling 54 TB in compressed fastq format. The SeqOthello index uses only 76.6 GB disk space, achieving a compression ratio of 700:1. Querying the index to assess the prevalence of 11,658 documented fusion events requires only five minutes on a standard desktop computer with 32 GB memory. This performance is orders of magnitude faster than the most-efficient existing fusion-detection algorithm, estimated to require 785 days of computation to process all TCGA data (methods).

SeqOthello queries on individual sequences as well as their constituent k-mers for their presence and absence in each experiment. The utility of SeqOthello's query result is demonstrated by the application of gene fusion survey, which accurately determines the tumor-specificity of individual gene fusion events without requiring downloading and re-analysis of raw sequencing data.

The simple query supported by SeqOthello is powerful, with myriad applications yet to be defined. One can use SeqOthello to assess the prevalence of clinically important features in different patient populations or to compare across different patient cohorts. Beyond transcripts, one can use SeqOthello to identify expressed regions by querying entire reference genomes. SeqOthello can be potentially leveraged on any form of next-generation sequencing data that can be translated to a k-mer occurrence matrix. We leave the definitions and demonstrations of these applications for future work.

6.7 Conclusion

In conclusion, SeqOthello is parameter-free, reference-free, and annotation-free. Its unbiased nature supports large-scale integrative and comparative studies, while its ultra-fast performance and undemanding system requirement render it appropriate for a wide variety of research investigators. SeqOthello will enable novel discoveries that would be otherwise unrealizable for individual research labs.

Chapter 7. Conclusion and Future Work

7.1 Dissertation summary

Research of computing technologies is facing emerging challenges in generating, transmitting, storing, and processing large-scale data. The work presented in this dissertation brings efficient algorithms and data structures for practical problems, by building real-world systems that tackle challenging issues of the distributed networking systems, big data, and bioinformatics.

Othello Hashing is an ultra-fast and memory-efficient key-value lookup method called. It fits the requirements of the core algorithms of many large-scale systems and big data applications. In this dissertation, I presented four applications developed using Othello Hashing with domain expertise in corresponding areas. They are the Concise forwarding information base, the SDLB software load balancer, the MetaOthello taxonomic classification tool, and the SeqOthello RNA-seq query engine. An extensive evaluation shows that these applications demonstrate significant performance improvement compared to existing approaches in the corresponding area.

7.2 Future work

Othello Hashing is an efficient algorithm that brings major performance improvement to the many areas in the cloud computing, networked systems, and bioinformatics area. The works presented in this dissertation initiates a series of research in multiple areas, in which there are still challenging problems remains to be solved. Also, there are additional research areas may benefit from the merits of Othello Hashing. Here, we outline some of the

research directions that we are interested to work on in the future.

Implementation of the Othello Hashing on programmable switching hardware

We acknowledge that prevalence of domain-specific language in network programming, especially in enabling customized data plane operations on hardware switches. Implementation of Othello Hashing on the hardware is a challenging but promising topic. How to utilize the programmability of the data plane remains an open problem for Othello Hashing. In specific, how to boost the performance of the data plane and how to utilize the platform-specific instructional optimization will be the first two challenges we need to tackle.

High-performance bioinformatics workflow processing

In the past years, the bioinformatics community has been using the Common Workflow Language for building pipelines. Integrating the tools presented in this paper with CWL interfaces would allow more users to have the access to these tools.

High throughput load balancer with consistency support for Layer 4 traffic

Network traffic sent to publicly known virtual IPs will be distributed by an L4LB to back-end servers. In this dissertation, we discussed the SDLB which is designed for mobile edge computing. For L4LB, the challenge remains in achieving faster packet processing, small memory cost, and no false hits. How to utilize the properties of alien key queries of Othello Hashing is also a challenging but promising issue.

Privacy-preserving storage system

The algorithmic tools in the homomorphic encryption field are usually slow and time-consuming. Although Othello is not designed for encryption computation, the very nature of Othello Hashing still shows other potentials for helping is that the query structure consists of two arrays. This may be helpful for constructing privacy-preserving systems.

Bibliography

[1] K. A. Wetterstrand, "DNA Sequencing Costs: Data from the NHGRI Large-Scale Genome Sequencing Program," pp. Date Accessed 2018–05–16. [Online]. Available: https://www.genome.gov/27541954/dna-sequencing-costs-data/

[2] M. Rouse and J. Burke, "Software-defined networking (SDN)," Tech. Rep. 2, 2013.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, mar 2008.

[4] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Functions Virtualization : Challenges and Opportunities for Innovations," *IEEE Communications Magazine*, pp. 90–97, 2014.

[5] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, jul 2013.

[6] S. Han, K. Jang, and L. Rizzo, "E2: A Framework for NFV Applications," in *SOSP*, 2015, pp. 121–136.

[7] H. Farhady, H. Lee, and A. Nakao, "Software-Defined Networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[8] J. Saltzer, "On the naming and binding of network destinations," *RFC 1498*, 1993.

[9] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish, "A layered naming architecture for the internet," in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4. ACM, 2004, pp. 343–352.

[10] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *Proc. of ACM CoNEXT*, 2009, pp. 313–324.

[11] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A Low Cost Netowrk Monitoring Framework for Software Defined Networks," in *Proc. of IEEE/IFIP NOMS*, 2014.

[12] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proc. of ACM SIGCOMM*, 2010.

[13] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal, and Others, "Mobile-edge computing introductory technical white paper," *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[14] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computingA key technology towards 5G," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[15] M. K. Susai, R. Sinha, D. S. Setia, and A. V. Soni, "Internet client-server multiplexer," *US Patent 6,954,780*, 2002.

[16] Loadba, "Zero Downtime from the load balancer experts - Loadbalancer.org," pp. Date Accessed 2018–05–25. [Online]. Available: https://www.loadbalancer.org/

[17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A Fast and Reliable Software Network Load Balancer," in *USENIX Symposium on Networked Systems Design and Implementation*, 2016, pp. 523–535.

[18] F. Crick, "Central dogma of molecular biology," *Nature*, vol. 227, no. 5258, pp. 561–563, 1970.

[19] National Centre for Biotechnology Information, "SRA: Sequence Read Archive," *NCBI Handout Series*, p. 4, 2015.

[20] TCGA, "The Cancer Genome Atlas," pp. Date Accessed: 2017–11–05.

[21] International Cancer Genome Consortium, "International Cancer Genome Consortium," pp. Date Accessed: 2017–11–05. [Online]. Available: http://icgc.org/

[22] T. Barrett, S. E. Wilhite, P. Ledoux, C. Evangelista, I. F. Kim, M. Tomashevsky, K. A. Marshall, K. H. Phillippy, P. M. Sherman, M. Holko, A. Yefanov, H. Lee, N. Zhang, C. L. Robertson, N. Serova, S. Davis, and A. Soboleva, "NCBI GEO: Archive for functional genomics data sets - Update," *Nucleic Acids Research*, vol. 41, no. D1, 2013.

[23] S. V. Angiuoli, M. Matalka, A. Gussman, K. Galens, M. Vangala, D. R. Riley, C. Arze, J. R. White, O. White, and W. F. Fricke, "CloVR: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing," *BMC Bioinformatics*, vol. 12, no. 1, p. 356, 2011.

[24] M. C. Schatz, B. Langmead, and S. L. Salzberg, "Cloud computing and the DNA data race," pp. 691–693, jul 2010.

[25] K. Krampis, T. Booth, B. Chapman, B. Tiwari, M. Bicak, D. Field, and K. E. Nelson, "Cloud BioLinux: Pre-configured and on-demand bioinformatics computing for the genomics community," *BMC Bioinformatics*, vol. 13, no. 1, p. 42, 2012.

[26] P. Amstutz, M. Crusoe, N. Tijanić, and B. Chapman, "Common Workflow Language Specifications, v1.0.2," pp. Date Accessed: 2018–05–01. [Online]. Available: https://www.commonwl.org/v1.0/

[27] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, nov 2004.

[28] C. Huttenhower and Human Microbiome Project Consortium, "Structure, function and diversity of the healthy human microbiome." *Nature*, vol. 486, no. 7402, pp. 207–14, 2012.

[29] J. C. Venter, K. Remington, J. F. Heidelberg, A. L. Halpern, D. Rusch, J. a. Eisen, D. Wu, I. Paulsen, K. E. Nelson, W. Nelson, D. E. Fouts, S. Levy, A. H. Knap, M. W. Lomas, K. Nealson, O. White, J. Peterson, J. Hoffman, R. Parsons, H. Baden-Tillson, C. Pfannkoch, Y.-H. Rogers, and H. O. Smith, "Environmental genome shotgun sequencing of the Sargasso Sea." *Science*, vol. 304, no. 5667, pp. 66–74, 2004.

[30] G. W. Tyson, J. Chapman, P. Hugenholtz, E. E. Allen, R. J. Ram, P. M. Richardson, V. V. Solovyev, E. M. Rubin, D. S. Rokhsar, and J. F. Banfield, "Community structure and metabolism through reconstruction of microbial genomes from the environment." *Nature*, vol. 428, no. 6978, pp. 37–43, 2004.

[31] T. Barrett, D. B. Troup, S. E. Wilhite, P. Ledoux, C. Evangelista, I. F. Kim, M. Tomashevsky, K. A. Marshall, K. H. Phillippy, P. M. Sherman, R. N. Muertter, M. Holko, O. Ayanbule, A. Yefanov, and A. Soboleva, "NCBI GEO: Archive for functional genomics data sets-10 years on," *Nucleic Acids Research*, vol. 39, no. SUPPL. 1, 2011.

[32] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "A concise forwarding information base for scalable and fast name lookups," in *Proceedings - International Conference on Network Protocols, ICNP*, vol. 2017-Octob. IEEE, oct 2017, pp. 1–10.

[33] Y. Yu, X. Li, and C. Qian, "SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing," in *Proceedings of the Workshop on Mobile Edge Communications (MECOMM)*. ACM Press, 2017, pp. 55–60.

[34] X. Liu, Y. Yu, J. J. Liu, C. Qian, J. J. Liu, C. F. Elliott, C. Qian, and J. J. Liu, "A Novel Data Structure to Support Ultra-fast Taxonomic Classification of Metagenomic Sequences with k-mer Signatures," in *Bioinformatics*, vol. 34, no. 1. Oxford University Press, jul 2017, pp. 171–178.

[35] Y. Yu, J. J. Liu, X. Liu, Y. Zhang, E. Magner, C. Qian, and J. J. Liu, "SeqOthello: Query over RNA-seq experiments at scale," *bioRxiv*, p. 258772, feb 2018.

[36] Z. J. Czech, G. Havas, and B. S. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, vol. 43, no. 5, pp. 257–264, 1992.

[37] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A Family of Perfect Hashing Methods," *The Computer Journal*, vol. 39, no. 6, pp. 547–554, jun 1996.

[38] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Monotone minimal perfect hashing: searching a sorted table with O (1) accesses," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM. Philadelphia, PA: Society for Industrial and Applied Mathematics, jan 2009, pp. 785–794.

[39] P. Woelfel, "Maintaining External Memory Efficient Hash Tables," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, Berlin, Heidelberg, 2006, pp. 508–519.

[40] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 738–761, aug 1994.

[41] R. J. Cichelli and R. J., "Minimal perfect hash functions made simple," *Communications of the ACM*, vol. 23, no. 1, pp. 17–19, jan 1980.

[42] F. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *Algorithms and Data Structures*, pp. 139–150, 2007.

[43] F. C. Botelho, R. Pagh, and N. Ziviani, "Practical perfect hashing in nearly optimal space," *Information Systems*, vol. 38, no. 1, pp. 108–131, 2013.

[44] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 30–39, 2004.

[45] D. Charles and K. Chellapilla, "Bloomier filters: A second look," in *Lecture Notes in Computer Science*, vol. 5193 LNCS, jul 2008, pp. 259–270.

[46] S. Iwata, "The Othello game on an nxn board is PSPACE-complete," *Theoretical Computer Science*, vol. 123, pp. 329–340, 1994.

[47] Wikipedia, "Reversi," pp. Date Accessed 2018–05–01. [Online]. Available: https://en.wikipedia.org/wiki/Reversi

[48] F. C. Botelho, N. Wormald, and N. Ziviani, "Cores of random r-partite hypergraphs," *Information Processing Letters*, vol. 112, no. 8-9, pp. 314–319, 2012.

[49] L. Devroye and P. Morin, "Cuckoo hashing: Further analysis," pp. 215–219, may 2003.

[50] S. Janson and M. J. Luczak, "Susceptibility in subcritical random graphs," *Journal of Mathematical Physics*, vol. 49, no. 12, p. 125207, 2008.

[51] S. Janson and O. Riordan, "Susceptibility in inhomogeneous random graphs," *Electronic Journal of Combinatorics*, vol. 19, no. 1, pp. 31–, 2012.

[52] B. Bollobás, S. Janson, and O. Riordan, "The phase transition in inhomogeneous random graphs," *Random Structures and Algorithms*, vol. 31, no. 1, pp. 3–122, aug 2007.

[53] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Anderson, "Scalable, High Performance Ethernet Forwarding with CuckooSwitch," in *Proc. of ACM CoNEXT*, 2013.

[54] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, may 2004.

[55] D. Zhou, B. Fan, H. Lim, D. G. Anderson, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh, "Scaling Up Clustered Network Appliances with ScaleBricks," in *Proc. of ACM SIGCOMM*, 2015.

[56] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 3—-14, 2008.

[57] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proceedings of ACM SIGCOMM*, 2009.

[58] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," in *Proceedings of ACM CoNEXT*, 2012.

[59] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, "Host identity protocol," Tech. Rep., 2008.

[60] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "Mobilityfirst: a robust and trustworthy mobility-centric architecture for the future internet," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 16, no. 3, pp. 2–13, 2012.

[61] D. G. Anderson, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, S. Shenker, D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable internet protocol," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 339–350.

[62] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter, "Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture," in *Proceedings of ACM/IEEE ANCS*, 2015.

[63] E. Kohler, "The Click Modular Router," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

[64] Intel, "Data Plane Development Kit," pp. Date Accessed: 2015–08–01, 2014. [Online]. Available: https://dpdk.org/

[65] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking For Networks," in *Proc. of USENIX NSDI*, 2012.

[66] V. Jacobson, Smetters, D. K., J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. of ACM CoNEXT*, 2009.

[67] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh, "Named data networking (ndn) project," *NDN Tech. report ndn-0001*, 2010.

[68] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, "ROFL," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, p. 363, aug 2006.

[69] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy, "Scalable routing on flat names," in *Conference on emerging Networking EXperiments and Technologies CoNEXT*. ACM Press, 2010, p. 1.

[70] C. Qian and S. Lam, "ROME: Routing On Metropolitan-scale Ethernet ," in *Proceedings of IEEE ICNP*, 2012.

[71] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 135–146.

[72] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP Lookup Performance with FIB Explosion," in *Proc. of the ACM SIGCOMM*, 2014.

[73] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 57–70.

[74] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A programmable, protocol-independent software switch," *2016 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2016*, pp. 525–538, 2016.

[75] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, and Others, "Wire Speed Name Lookup: A GPU-based Approach." in *NSDI*, 2013, pp. 199–212.

[76] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A Shifting Framework for Set Queries," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3116–3131, jan 2017.

[77] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, may 2015.

[78] CAIDA, "The CAIDA UCSD Anonymized Internet Traces 2013 - 2014. Mar." pp. Date Accessed 2016–05–02. [Online]. Available: http://www.caida.org/data/passive/passive_2013_dataset.xml

[79] S. Jain and Others, "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proceedings of ACM Sigcomm*, 2013.

[80] L. M. Vaquero and L. Rodero-Merino, "Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, oct 2014.

[81] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[82] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*. IEEE, 2015, pp. 73–78.

[83] E. Borcoci, "Fog Computing, Mobile Edge Computing, Cloudlets - which one?" in *SoftNet Conference*, 2016.

[84] S. Yi, Z. Qin, and Q. Li, "Security and Privacy Issues of Fog Computing: A Survey," in *International Conference on Wireless Algorithms, Systems, and Applications*, 2015, pp. 685–695.

[85] Y. Yu, Q. Chen, and X. Li, "Distributed Collaborative Monitoring in Software Defined Networks," *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN*, pp. 85–90, 2014.

[86] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[87] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, 2013.

[88] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. of ACM MobiSys*, 2010.

[89] HAProxy, "HAProxy - the Reliable, High Performance TCP/HTTP Load Balancer," pp. Date Accessed 2017–07–06. [Online]. Available: http://www.haproxy.org/

[90] LVS, "the Linux Virtual Server project," pp. Date Accessed 2017–10–15. [Online]. Available: http://www.linuxvirtualserver.org/

[91] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[92] T. Willhalm, R. Dementiev, and P. Fay, "Intel Performance Counter Monitor," pp. Date Accessed 2017–06–20. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[93] X. Li and C. Qian, "Low-Complexity Multi-Resource Packet Scheduling for Network Function Virtualization," in *Proceedings of IEEE INFOCOM*, 2015.

[94] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack, "Analysis of LAS scheduling for job size distributions with high variance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 218–228, 2003.

[95] X. Li and C. Qian, "An NFV Orchestration Framework for Interference-free Policy Enforcement," in *Proc. of IEEE ICDCS*, 2016.

[96] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[97] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool." *Journal of molecular biology*, vol. 215, no. 3, pp. 403–10, 1990.

[98] D. H. Huson, A. F. Auch, J. Qi, and S. C. Schuster, "MEGAN analysis of metagenomic data," *Genome Research*, vol. 17, no. 3, pp. 377–386, mar 2007.

[99] A. Brady and S. L. Salzberg, "Phymm and PhymmBL: metagenomic phylogenetic classification with interpolated Markov models," *Nature Methods*, vol. 6, no. 9, pp. 673–676, sep 2009.

[100] G. L. Rosen, E. R. Reichenberger, and A. M. Rosenfeld, "NBC: The naïve Bayes classification tool webserver for taxonomic classification of metagenomic reads," *Bioinformatics*, vol. 27, no. 1, pp. 127–129, jan 2011.

[101] D. Kim, L. Song, F. P. Breitwieser, and S. L. Salzberg, "Centrifuge: rapid and sensitive classification of metagenomic sequences," *Genome Research*, vol. 26, no. 12, pp. 1721—-1729, 2016.

[102] P. Menzel and A. Krogh, "Fast and sensitive taxonomic classification for metagenomics with Kaiju," *Nature communications*, vol. 7, p. 11257, 2016.

[103] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen, "Scalable metagenomic taxonomy classification using a reference genome database." *Bioinformatics (Oxford, England)*, vol. 29, no. 18, pp. 2253–60, sep 2013.

[104] D. E. Wood and S. L. Salzberg, "Kraken: ultrafast metagenomic sequence classification using exact alignments." *Genome biology*, vol. 15, no. 3, p. R46, 2014.

[105] S. Lindgreen, K. L. Adair, and P. P. Gardner, "An evaluation of the accuracy and speed of metagenome analysis tools," *Scientific Reports*, vol. 6, p. 19233, 2016.

[106] C. F. Davenport, J. Neugebauer, N. Beckmann, B. Friedrich, B. Kameri, S. Kokott, M. Paetow, B. Siekmann, M. Wieding-Drewes, M. Wienhöfer, S. Wolf, B. Tümmler, V. Ahlers, and F. Sprengel, "Genometa - A fast and accurate classifier for short metagenomic shotgun reads," *PLoS ONE*, vol. 7, no. 8, 2012.

[107] T. A. K. Freitas, P.-E. Li, M. B. Scholz, and P. S. G. Chain, "Accurate read-based metagenome characterization using a hierarchical suite of unique signatures," *Nucleic Acids Research*, p. gkv180, 2015.

[108] D. H. Huson, S. Mitra, H.-J. Ruscheweyh, N. Weber, and S. C. Schuster, "Integrative analysis of environmental sequences using MEGAN4," *Genome Research*, vol. 21, no. 9, pp. 1552–1560, 2011.

[109] F. Meyer, D. Paarmann, M. D'Souza, R. Olson, E. Glass, M. Kubal, T. Paczian, A. Rodriguez, R. Stevens, A. Wilke, J. Wilkening, R. Edwards, J. Venter, K. Remington, J. Heidelberg, A. Halpern, D. Rusch, J. Eisen, D. Wu, I. Paulsen, K. Nelson, W. Nelson, G. Tyson, J. Chapman, P. Hugenholtz, E. Allen, R. Ram, P. Richardson, V. Solovyev, E. Rubin, D. Rokhsar, J. Banfield, S. Huse, J. Huber, H. Morrison, M. Sogin, D. Welch, R. Overbeek, T. Begley, R. Butler, J. Choudhuri, N. Diaz, H.-Y. Chuang, M. Cohoon, V. de Crécy-Lagard, T. Disz, R. Edwards, L. McNeil, C. Reich, R. Aziz, D. Bartels, M. Cohoon, T. Disz, R. Edwards, S. Gerdes, K. Hwang, M. Kubal, M. Margulies, M. Egholm, W. Altman, S. Attiya, J. Bader, L. Bemben, J. Berka, M. Braverman, Y. Chen, Z. Chen, R. Aziz, D. Bartels, A. Best, M. DeJongh, T. Disz, R. Edwards, K. Formsma, S. Gerdes, E. Glass, M. Kubal, D. Field, N. Morrison, J. Selengut, P. Sterk, S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, D. Lipman, T. Jarvie, T. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. Brodie, K. Keller, T. Huber, D. Dalevil, P. Hu, G. Andersen, J. Cole, B. Chai, R. Farris, Q. Wang, J. Wuyts, Y. de Peer, T. Winkelmans, R. D. Wachter, R. Leplae, A. Hebrant, S. Wodak, A. Toussaint, F. Meyer, R. Overbeek, A. Rodriquez, S. Tringe, C. von Mering, A. Kobayashi, A. Salamov, K. Chen, H. Chang, M. Podar, J. Short, E. Mathur, J. Detter, B. Rodriguez-Brito, F. Rohwer, R. Edwards, A. McHardy, H. Martin, A. Tsirigos, P. Hugenholtz, I. Rigoutsos, R. Edwards, B. Rodriguez-Brito, L. Wegley, M. Haynes, M. Breitbart, D. Peterson, M. Saar, S. Alexander, E. Alexander, F. Rohwer, N. Fierer, M. Breitbart, J. Nulton, P. Salamon, C. Lozupone, R. Jones, M. Robeson, R. Edwards, B. Felts, S. Rayhawk, L. Wegley, R. Edwards, B. Rodriguez-Brito, H. Liu, F. Rohwer, X. Mou, R. Edwards, R. Hodson, M. Moran, E. Dinsdale, R. Edwards, D. Hall, F. Angly, M. Breitbart, J. Brulc, M. Furlan, C. Desnues, M. Haynes, L. Li, L. Krause, N. Diaz, D. Bartels, R. Edwards, A. Puhler, F. Rohwer, F. Meyer, J. Stoye, F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg, J. Quackenbush, and F. Rohwer, "The metagenomics RAST server a public resource for the automatic phylogenetic and functional analysis of metagenomes," *BMC Bioinformatics*, vol. 9, no. 1, p. 386, sep 2008.

[110] J. Dröge, I. Gregor, and A. C. McHardy, "Taxator-tk: Precise taxonomic assignment of metagenomes by fast approximation of evolutionary neighborhoods," *Bioinformatics*, vol. 31, no. 6, pp. 817–824, 2015.

[111] N. Segata, L. Waldron, A. Ballarini, V. Narasimhan, O. Jousson, and C. Huttenhower, "Metagenomic microbial community profiling using unique clade-specific marker genes." *Nature methods*, vol. 9, no. 8, pp. 811–4, 2012.

[112] B. Liu, T. Gibbons, M. Ghodsi, and M. Pop, "MetaPhyler: Taxonomic profiling for metagenomic sequences," in *Proceedings - 2010 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2010*, 2010, pp. 95–100.

[113] S. Sunagawa, D. R. Mende, G. Zeller, F. Izquierdo-Carrasco, S. a. Berger, J. R. Kultima, L. P. Coelho, M. Arumugam, J. Tap, H. B. Nielsen, S. Rasmussen, S. Brunak, O. Pedersen, F. Guarner, W. M. de Vos, J. Wang, J. Li, J. Dore, S. D. Ehrlich, A. Stamatakis, and P. Bork, "Metagenomic species profiling using universal phylogenetic marker genes," *Nat Methods*, vol. 10, no. 12, pp. 1196–1199, 2013.

[114] J. G. Caporaso, J. Kuczynski, J. Stombaugh, K. Bittinger, F. D. Bushman, E. K. Costello, N. Fierer, A. G. Peña, J. K. Goodrich, J. I. Gordon, G. A. Huttley, S. T. Kelley, D. Knights, J. E. Koenig, R. E. Ley, C. A. Lozupone, D. McDonald, B. D. Muegge, M. Pirrung, J. Reeder, J. R. Sevinsky, P. J. Turnbaugh, W. A. Walters, J. Widmann, T. Yatsunenko, J. Zaneveld, and R. Knight, "QIIME allows analysis of high-throughput community sequencing data." *Nature methods*, vol. 7, no. 5, pp. 335–6, 2010.

[115] M. Pertea, D. Kim, G. M. Pertea, J. T. Leek, and S. L. Salzberg, "Transcript-level expression analysis of RNA-seq experiments with HISAT, StringTie and Ballgown," *Nature protocols*, vol. 11, no. 9, pp. 1650–1667, 2016.

[116] N. R. NCBI Resource Coordinators, A. Acland, R. Agarwala, T. Barrett, J. Beck, D. A. Benson, C. Bollin, E. Bolton, S. H. Bryant, K. Canese, D. M. Church, K. Clark, M. DiCuccio, I. Dondoshansky, S. Federhen, M. Feolo, L. Y. Geer, V. Gorelenkov, M. Hoeppner, M. Johnson, C. Kelly, V. Khotomlianski, A. Kimchi, M. Kimelman, P. Kitts, S. Krasnov, A. Kuznetsov, D. Landsman, D. J. Lipman, Z. Lu, T. L. Madden, T. Madej, D. R. Maglott, A. Marchler-Bauer, I. Karsch-Mizrachi, T. Murphy, J. Ostell, C. O'Sullivan, A. Panchenko, L. Phan, D. P. K. D. Pruitt, W. Rubinstein, E. W. Sayers, V. Schneider, G. D. Schuler, E. Sequeira, S. T. Sherry, M. Shumway, K. Sirotkin, K. Siyan, D. Slotta, A. Soboleva, V. Soussov, G. Starchenko, T. A. Tatusova, B. W. Trawick, D. Vakatov, Y. Wang, M. Ward, W. J. Wilbur, E. Yaschenko, and K. Zbicz, "Database resources of the National Center for Biotechnology Information." *Nucleic acids research*, vol. 42, no. Database issue, pp. D7–17, jan 2014.

[117] R. Petryszak, T. Burdett, B. Fiorelli, N. A. Fonseca, M. Gonzalez-Porta, E. Hastings, W. Huber, S. Jupp, M. Keays, N. Kryvych, J. McMurry, J. C. Marioni, J. Malone, K. Megy, G. Rustici, A. Y. Tang, J. Taubert, E. Williams, O. Mannion, H. E. Parkinson, and A. Brazma, "Expression Atlas update–a database of gene and transcript expression from microarray- and sequencing-based functional genomics experiments." *Nucleic acids research*, vol. 42, no. Database issue, pp. D926–32, jan 2014.

[118] L. Collado-Torres, A. Nellore, K. Kammers, S. E. Ellis, M. A. Taub, K. D. Hansen, A. E. Jaffe, B. Langmead, and J. T. Leek, "Reproducible RNA-seq analysis using recount2," pp. 319–321, 2017.

[119] A. Nellore, A. E. Jaffe, J.-P. P. Fortin, J. Alquicira-Hernández, L. Collado-Torres, S. Wang, R. A. Phillips III, N. Karbhari, K. D. Hansen, B. Langmead, J. T. Leek, R. A. Phillips, N. Karbhari, K. D. Hansen, B. Langmead, and J. T. Leek, "Human splicing diversity and the extent of unannotated splice junctions across human RNA-seq samples on the Sequence Read Archive," *Genome Biology*, vol. 17, no. 1, p. 266, 2016.

[120] B. Solomon and C. Kingsford, "Fast search of thousands of short-read sequencing experiments," *Nature Biotechnology*, vol. 34, no. 3, pp. 300–302, feb 2016.

[121] ——, "Improved search of large transcriptomic sequencing databases using split sequence bloom trees," vol. 10229 LNCS, pp. 257–271, dec 2017.

[122] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev, "Allsome sequence bloom trees," in *Lecture Notes in Computer Science*, vol. 10229 LNCS. Cold Spring Harbor Laboratory, dec 2017, pp. 272–286.

[123] S. Tarkoma, C. C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[124] G. Holley, R. Wittler, and J. Stoye, "Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage," *Algorithms for Molecular Biology*, vol. 11, no. 1, p. 3, dec 2016.

[125] D. D. Dolle, Z. Liu, M. Cotten, J. T. Simpson, Z. Iqbal, R. Durbin, S. A. McCarthy, and T. M. Keane, "Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes." *Genome research*, vol. 27, no. 2, pp. 300–309, feb 2017.

[126] 1000 Genomes Project Consortium, A. Auton, L. D. Brooks, R. M. Durbin, E. P. Garrison, H. M. Kang, J. O. Korbel, J. L. Marchini, S. McCarthy, G. A. McVean, and G. R. Abecasis, "A global reference for human genetic variation." *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[127] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. July 1928, pp. 379–423, 1948.

[128] M. Borda, "Improved Docking of Polypeptides with Glide," *Journal of Chemical Information and Modeling*, vol. 53, no. 9, pp. 1689–1699, 2011.

[129] G. Marçais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, mar 2011.

[130] J. Harrow, A. Frankish, J. M. Gonzalez, E. Tapanari, M. Diekhans, F. Kokocinski, B. L. Aken, D. Barrell, A. Zadissa, S. Searle, I. Barnes, A. Bignell, V. Boychenko, T. Hunt, M. Kay, G. Mukherjee, J. Rajan, G. Despacio-Reyes, G. Saunders, C. Steward, R. Harte, M. Lin, C. Howald, A. Tanzer, T. Derrien, J. Chrast, N. Walters,

S. Balasubramanian, B. Pei, M. Tress, J. M. Rodriguez, I. Ezkurdia, J. Van Baren, M. Brent, D. Haussler, M. Kellis, A. Valencia, A. Reymond, M. Gerstein, R. Guigó, and T. J. Hubbard, "GENCODE: The reference human genome annotation for the ENCODE project," *Genome Research*, vol. 22, no. 9, pp. 1760–1774, 2012.

[131] K. Yoshihara, Q. Wang, W. Torres-Garcia, S. Zheng, R. Vegesna, H. Kim, and R. G. Verhaak, "The landscape and therapeutic relevance of cancer-associated transcript fusions," *Oncogene*, vol. 34, no. 37, pp. 4845–4854, sep 2015.

[132] W. Torres-García, S. Zheng, A. Sivachenko, R. Vegesna, Q. Wang, R. Yao, M. F. Berger, J. N. Weinstein, G. Getz, and R. G. Verhaak, "PRADA: Pipeline for RNA sequencing data analysis," *Bioinformatics*, vol. 30, no. 15, pp. 2224–2226, aug 2014.

Vita

- **Personal Information**
  - **Name:** Ye Yu
  - **Birth place:** Qingdao, China.
- **Education**
  - Beihang Univeristy (Beijing University of Astronautics and Aeronautics), B. E. awarded in May 2013, Beijing, China
- **Employment History**
  - Software Development Engineer Intern. Facebook, Inc. Menlo Park, CA, USA. May 2017 - Aug 2017.
  - Research Intern. Ericsson. San Jose, CA, USA. Jun 2015 - Aug 2015.
  - Graduate Research / Teaching Assistant. Univeristy of Kentucky. Lexington, KY, USA. Aug. 2013 - May 2018.
- **Scholastic and Professional Awards**
  - Best Ph.D. Student, Department of Computer Science, Univeristy of Kentucky. 2018.
  - Student Travel Awards: IEEE ICNP 2014, 2017. IEEE CloudCom, 2016. ACM SIGCOMM, 2017. ACM SIGMETRICS 2017.
  - MSRA Undergraduate Scholarship, Microsoft Research Asia, 2012.
  - Scholarship by Qian Changzhao and Shen Xingyuan, Beihang University, 2011.
  - Scholarship for disciplinary contest winners, Beihang Univeristy, 2010.
  - Scholarship for outstanding new students, Beihang University, 2009.
  - Silver Medal in the ACM International Collegiate Programming Contest (ICPC) Asia Regional Contest, Tianjin, China, 2010.
  - First place prize in the Olympiad in Informatics of Shandong Province, Jinan, China, 2008.
  - Second prize in the Shandong Province Middle School Mathematics Olympics, Jinan, China, 2007.

- **Publications**

1. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing
   Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang
   in **IEEE/ACM Transactions on Networking (ToN)**, 2018.
2. NetCP: Consistent, Non-interruptive and Efficient Checkpointing and Rollback of SDN
   Ye Yu, Ying Zhang, Wenfei Wu, and Chen Qian
   in **IEEE/ACM International Symposium on Quality of Service (IWQoS)**, 2018.
3. A Novel Data Structure to Support Ultra-fast Taxonomic Classification of Metagenomic Sequences with k-mer Signatures
   Ye Yu*, Xinan Liu* (Co-first authorship), James N. MacLeod, Chen Qian and Jinze Liu
   in **Bioinformatics**, Oxford Academic, 2017.
4. Practical Network-wide Packet Behavior Identification by AP Classifier
   Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S. Lam
   in **IEEE/ACM Transactions on Networking (ToN)**, 2017.
5. Space Shuffle: A Scalable, Flexible, and High-Bandwidth Data Center Network
   Ye Yu and Chen Qian
   in **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, 2016.
6. DiFS: Distributed Flow Scheduling for adaptive switching in FatTree data center networks
   Wenzhi Cui, Ye Yu, and Chen Qian
   In **Computer Networks**, 2016.
7. A Concise Forwarding Information Base for Scalable and Fast Name Switching
   Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang
   in **IEEE International Conference on Network Protocols** (**ICNP**), 2017.
8. A Fast, Small, and Dynamic Forwarding Information Base
   Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang
   in **ACM SIGMETRICS**, 2017. Extended abstract.
9. SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing
   Ye Yu, Xin Li, and Chen Qian
   in **ACM SIGCOMM Workshop on Mobile Edge Communications** (**MECOMM**), 2017
10. A Novel Data Structure to Support Ultra-fast Taxonomic Classification of Metagenomic Sequences with k-mer Signatures
    Ye Yu*, Xinan Liu*(Co-first authorship), James N. MacLeod, Chen Qian and Jinze Liu
    in **RECOMB Workshop on Massively Parallel Sequencing** (**RECOMB-Seq**), 2017.

11. An IoT Data Communication Framework for Authenticity and Integrity
Xin Li, Huazhe Wang, Ye Yu, and Chen Qian
in **ACM/IEEE International Conference on Internet-of-Things Design and Implementation** (**IoTDI**), 2017.

12. Failure-Resilient Routing for Server-Centric Data Center Networks with Random Topologies
Ye Yu and Chen Qian
in **IEEE International Conference on Cloud Computing Technology and Science** (**CloudCom**), 2016.

13. Garlic Cast: Lightweight and Decentralized Anonymous Content Sharing
Chen Qian, Junjie Shi, Zihao Yu, Ye Yu, and Sheng Zhong
in **IEEE International Conference on Parallel and Distributed Systems (ICPADS)**, 2016

14. FTDC: A Fault-Tolerant Server-Centric Data Center Network
Ye Yu and Chen Qian
in **IEEE/ACM International Symposium on Quality of Service** (**IWQoS**), Poster, 2016.

15. Practical Network-wide Packet Behavior Identification by AP Classifier
Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S. Lam
in **ACM Conference on emerging Networking Experiments and Technologies** (**CoNEXT**), 2015.

16. Distributed Collaborative Monitoring in Software Defined Networks
Ye Yu, Chen Qian, and Xin Li
in **ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking** (**HotSDN**), 2014.

17. Space Shuffle: A Scalable, Flexible, and High-Bandwidth Data Center Network
Ye Yu and Chen Qian
in **IEEE International Conference on Network Protocols** (**ICNP**), 2014.

18. NetCP: Multi-level Checkpointing and Rollback of Software Defined Non-interrupted Networks
Ye Yu, Chen Qian, Ying Zhang, Ravi Manghirmalani
in **IEEE International Conference on Network Protocols** (**ICNP**), Poster, 2014.