Fast Decisions in High-Speed Networking Devices

Yossi Kanizo
Fast Decisions in High-Speed Networking Devices

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Yossi Kanizo

Submitted to the Senate of the Technion — Israel Institute of Technology
Sh’vat, 5774 Haifa January, 2014
The research thesis was done under the supervision of Prof. Isaac Keslassy and Dr. David Hay in the department of computer science.

Acknowledgments

I want to thank my advisors, Isaac Keslassy and David Hay, for their academic guidance during this long and challenging journey. Your invaluable contribution has been a great influence to the way I approach research problems, and particularly, to the way I define them. Not only has it helped me to bring this journey to its successful end, but also, I believe, it will serve me well in my professional life.

Throughout this journey, I also have met other smart people whom I have had contributing discussions with. For these discussions, I would like to thank Hagit Attiya, Danny Raz, Jennifer Rexford, Anat Bremler-Barr, Eldar Fischer, Yuval Shavitt, and Seffi Naor. I also would like to thank the following co-authors: Alex Zlotnik and Ori Rottenstreich, who have also become true friends.

Last and most importantly, I would like to thank my family for their endless support and unconditional love. My parents, Sara and Avi Kanizo, and my dear brother Yoni Kanizo, none of this could have happened without you. This dissertation is dedicated to you.

The generous financial support of the Technion is gratefully acknowledged. In addition, the work was partly supported by the European Research Council Starting Grant n°210389, the European Research Council Starting Grant n°259085, the Alon Fellowship, the ATS-WD Career Development Chair, the Loewengart Research Fund, the Intel ICRI-CI Center, and the Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11).
Publication List

Papers covered in this dissertation


Other papers


Contents

Abstract 1

Abbreviations and Notations 2

1 Introduction 3
   1.1 Network Switches and Fast Decisions 3
   1.2 Tables in Switches 4
      1.2.1 Hash Table 4
      1.2.2 Content-Addressable Memory (CAM) and Ternary Content-Addressable Memory (TCAM) 5
   1.3 Specificity of Networking Devices 5
   1.4 Performance Metrics 6
      1.4.1 Speed 6
      1.4.2 Space Requirement 7
      1.4.3 Energy Considerations 7
   1.5 Dissertation Overview and Main Contributions 7
      1.5.1 The Static Case 8
      1.5.2 The Dynamic Case 8
      1.5.3 The Balancing Problem and Its Application to Bloom Filters 9
      1.5.4 Combined Memory Model via General Lower Bound 10
      1.5.5 Spreading Tables Across The Network 10

2 Related Work 12
   2.1 Multiple-Choice Hash Tables 12
      2.1.1 d-random and d-left 12
      2.1.2 Multi-Level Hash Table 13
      2.1.3 Hash Table Schemes With Moves 13
      2.1.4 Hash Table With Deletions 14
      2.1.5 Hash Table With a Combined Memory 14
   2.2 Bloom Filter And Its Variants 15
      2.2.1 Bloom Filter 15
      2.2.2 Blocked Bloom Filter 16
### Contents

2.2.3 Bloom Filters via $d$-left Hashing and Dynamic Bit Reassignment . . 17
2.2.4 Energy-Efficient Bloom Filters . . . . . . . . . . . . . . . . . . . . . 17
2.3 Load Balancing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.4 Random Graph Matching Problem . . . . . . . . . . . . . . . . . . . . . .. 18
2.5 Software Defined Networks and Network Coordination . . . . . . . . . . 18

#### 3 Model Definition

4 Static Table Model 23
4.1 Summary of Our Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 Overflow Lower Bound . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   4.3.1 The Cloning Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   4.3.2 Identical Hash Function Distributions . . . . . . . . . . . . . . . . . . 26
   4.3.3 Multiple Hash Function Distributions . . . . . . . . . . . . . . . . . . 30
4.4 SIMPLE - A Single-Choice Hashing Scheme . . . . . . . . . . . . . . . . 32
   4.4.1 Description by Differential Equations . . . . . . . . . . . . . . . . . . 33
   4.4.2 Optimality of the SIMPLE Scheme . . . . . . . . . . . . . . . . . . . 34
   4.4.3 Case Study - A SIMPLE $\langle 1, 1, c, 1 \rangle$ Hashing Scheme . . . . 35
   4.4.4 Simulation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.5 GREEDY - A Multiple-Choice Hashing Scheme . . . . . . . . . . . . . . 37
   4.5.1 Description by Differential Equations . . . . . . . . . . . . . . . . . . 37
   4.5.2 Optimality of the GREEDY Scheme . . . . . . . . . . . . . . . . . . . 38
   4.5.3 Case Study - A GREEDY $\langle a, 2, c, 1 \rangle$ Hashing-Scheme . . . . 39
4.6 The Multi-Level Hash Table (MHT) Scheme . . . . . . . . . . . . . . . . 40
   4.6.1 Description by Differential Equations . . . . . . . . . . . . . . . . . . 40
   4.6.2 Reduction to the SIMPLE Scheme . . . . . . . . . . . . . . . . . . . . 41
   4.6.3 Optimality of the MHT Scheme . . . . . . . . . . . . . . . . . . . . . . 42
   4.6.4 Case Study - A MHT $\langle a, 2, 1, 1 \rangle$ Hashing-Scheme . . . . . 43
4.7 Dealing With Non-Unique Elements . . . . . . . . . . . . . . . . . . . . . . 44
   4.7.1 Model Given the Arrival Order of Elements . . . . . . . . . . . . . . . 45
   4.7.2 Model Given the Instances Histogram . . . . . . . . . . . . . . . . . . 46
4.8 Comparative Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
   4.8.1 Lower bound vs. GREEDY and MHT . . . . . . . . . . . . . . . . . . . 47
   4.8.2 Effects of the System Parameters . . . . . . . . . . . . . . . . . . . . . 48
   4.8.3 Trace-Driven Simulation With Non-Unique Elements . . . . . . . . . 49

5 Dynamic Table Model 52
5.1 Performance Degradation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
   5.1.1 Simulation results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
   5.1.2 Intuition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.2 Summary of Our Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.3 Problem Statement ............................................. 56
  5.3.1 Terminology and Notations .............................. 56
  5.3.2 Input Models ............................................ 56
5.4 A Single-Choice Hashing Scheme .............................. 58
5.5 Overflow Lower Bound ...................................... 63
5.6 Lower Bound with Multiple Hash-Function Distributions .... 66
5.7 A Multiple-Choice Hashing Scheme ............................ 67
5.8 Moving Back Elements ...................................... 69
  5.8.1 Description .................................. 70
  5.8.2 Analysis ...................................... 70
5.9 Experimental Results ...................................... 71
  5.9.1 Simulations ................................... 71
  5.9.2 Experiments Using Real-Life Traces .................. 73
  5.9.3 Experiments Using an On-off Arrival Model ......... 74
6 Balancing Table Buckets .................................... 76
  6.1 Bloom Filter Inefficiency ................................ 76
  6.2 Summary of Our Results .................................. 77
  6.3 Balanced Bloom Filters .................................. 78
  6.4 Problem Statement ...................................... 79
  6.5 Theoretical Lower Bounds ................................ 80
  6.6 SINGLE-B - A Single-Choice Balancing Scheme ........ 85
    6.6.1 Description by Differential Equations .......... 85
    6.6.2 Optimality Result ................................ 86
  6.7 SEQUENTIAL-B - A Multiple-Choice Balancing Scheme ... 88
    6.7.1 Description by Differential Equations .......... 88
    6.7.2 Optimality Result ................................ 89
  6.8 The Multi-Level Hash Table (MHT-B) Balancing Scheme ... 90
    6.8.1 Description by Differential Equations .......... 91
  6.9 Optimality Result ...................................... 93
  6.10 Comparative Evaluation and Analysis ..................... 94
  6.11 Analysis of the Balanced Bloom Filter ................. 94
    6.11.1 Balanced Bloom Filter with Overflow List ........ 95
    6.11.2 Balanced Bloom Filter Without Overflow List .... 97
  6.12 Trace-Driven Experiments ................................ 97
  6.13 Additional Application: Variance of the Query-Time in Chain-Based Hash-Tables ........................................... 98
7 Tables in Combined Memories ................................ 100
  7.1 Summary of Our Results .................................. 101
  7.2 Bipartite Graph Model .................................. 103
List of Figures

1.1 Off-chip hash table. The dashed rectangle delimits the chip’s area and illustrates its pin in/out bandwidth. .................. 6
2.1 Illustration of Bloom filter [13] .......................... 15
2.2 Illustration of Blocked Bloom filter ......................... 16
3.1 Illustration of the hashing model .......................... 21
3.2 Illustration of the mht scheme ............................. 21
4.1 $\gamma_{lb}(a)$ as a function of $c$, given $a = 1.2$ .................. 30
4.2 $\alpha_{lb}(\gamma)$ as a function of $c$, given $\gamma = 0.01$ ............... 30
4.3 An example of two different distributions (probability mass functions) $f_1$ and $f_2$ that reach the lower bound with $k_1 = \frac{1}{3}$ and $k_2 = \frac{2}{3}$ .................. 32
4.4 Illustration of the SIMPLE scheme .......................... 32
4.5 Model and simulation results for the SIMPLE scheme given load $c = 1$, bucket size $h = 3$, and memory size $m = 10,000$ .............. 36
4.6 Illustration of the GREEDY scheme .......................... 37
4.7 Overflow fraction as a function of $a$ with $d = 4$, $h = 4$, $c = 1$. OP($a$) denotes the optimal partition of mht for $a$ as obtained by Theorem 7 .................. 47
4.8 Cut-off points of GREEDY and MHT schemes, and the corresponding overflow fraction, as a function of the load $c$, with bucket size $h = 4$ and $d = 4$ hash functions .......................... 48
4.9 Cut-off points of GREEDY and MHT schemes, and the corresponding overflow fraction, as a function of the maximum number of memory accesses $d$, with load of $c = 0.9$ and bucket size $h = 4$ ................. 49
4.10 Average number of memory accesses with non-unique elements ....... 50
4.11 Overflow fraction with non-unique elements .................. 50
5.1 Average overflow fraction with 2 hash functions and bucket size 1, using both the static and the dynamic model .......................... 53
5.2 An example demonstrating the degradation of performance in dynamic hash tables .......................... 53
5.3 Overflow fraction as a function of the number of buckets $m$ ............. 60
5.4 Expected overflow fraction as a function of the average memory access rate $a$. 66
5.5 Illustration of the MULTIPLE scheme ................................. 68
5.6 Expected overflow fraction as a function of $a$ with $d = 4$, $h = 4$, $c = 1$ .... 72
5.7 M-B with MULTIPLE scheme, for $h = 4$, $d = 2$ and different loads ...... 72
5.8 Expected overflow fraction of the proposed moving-back (M-B) scheme (via simulations) ........................................... 73
5.9 Experiment using real-life traces and hash functions with SINGLE and MULTIPLE (d=2). .................................................. 73
5.10 The M-B and the hints-based schemes, for $h = 4$, $d = 1$ and different loads. 74
5.11 Marginal overflow fraction of 100 on-off flows with $m = 500$, $h = 1$ and $d = 2$ 74

6.1 Illustration of a Balanced Bloom filter implementation based on MHT-B, with three subtables. .................................................. 78
6.2 The probability density function of the distribution $P_{lb}$ over the bucket occupancies with load $r = 8$, overflow fraction $\gamma = 0$, and different values of average access rate $a$. ........................................... 84
6.3 Simulation vs. analytical model for SINGLE-B with $r = 2.5$, $p = 0.5$ and $\bar{h} = 2$ 87
6.4 The overflow fraction $\gamma$ induced by applying the SEQUENTIAL-B and MHT-B schemes with worst-case memory access rate $d = 3$. ......................... 95
6.5 False positive rates of different Bloom filter schemes with memory block size of $B = 256$ bits and variable load $r$. SINGLE-B and MHT-B use overflow fraction $\gamma \approx 0.5\%$. .................................................. 96
6.6 Trace-driven experiments of the MHT-B-based Balanced Bloom filter scheme different values of $r$. ................................. 98

7.1 Example of bipartite graph with left-side vertex degree 2 ................. 105
7.2 the Lambert-$W$ function .................................................... 111
7.3 Expected maximum matching size for various values of $n$ and $m$ (normalized by $n$). .......................................................... 127
7.4 Limit expected normalized maximum matching size $\psi$ for various values of load $c$. ....................................................... 127
7.5 Limit expected normalized maximum matching size $\psi$ as a function of the average number of choices $a$, for various values of the load $c$. ................. 128
7.6 limit expected normalized maximum matching size $\psi_{a_1}$ as a function of the partition parameter $\alpha_1$, for various values of the load $c$. ................ 128
7.7 comparison of the simulation results and the theoretical upper bound for the normalized expected maximum matching size, as a function of the number of choices $d$ in any multiple-choice hashing scheme. The load is $c = 1$. .... 129
7.8 Experiment using real-life traces and hash functions of the normalized number of elements in the DRAM, and comparison to the theoretical model. .... 130
7.9 Expected access throughput given that the off-chip memory is \( b = 5 \) times slower than the on-chip memory ........................................ 130

7.10 Ratio of the throughput in the cases of using maximum size matching (MSM) with \( d = 3 \) and \( d = 4 \) by the throughput of optimally-partitioned maximum size matching based hashing with \( d = 2 \). A comparison with \( d \)-left hashing scheme is also plotted. ........................................ 131

8.1 (a) A common setting in which tables are installed at the network ingress nodes; (b) The result of applying Palette. Tables are decomposed into smaller subtables of different types (a.k.a. colors), which are then distributed across the network. A packet along each path meets each type of subtable at least once. ........................................ 133

8.2 An example rule-set of an SDN table. ........................................ 137

8.3 The dependency graph and cut of the table in Fig. 8.2. ................. 140

8.4 Illustration of the model. Network \( G = \langle V, E \rangle \) has vertex set \( V = \{v_1, \ldots, v_4\} \) for the switches, and an edge set \( E = \{e_1, \ldots, e_3\} \) for the links. There are three paths in the path set \( P = \{p_1, p_2, p_3\} \), and, for example, \( S(p_1) = \{v_1, v_2\} \) and \( L(p_1) = \{e_1\} \). Finally, \( G|_P \) is the same as \( G \) but without the link \( e_3 \), which does not belong to any of the paths. ............ 142

8.5 A network with no valid coloring to the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM with \( w = 2 \) ........................................ 149

8.6 Evaluation of the quality metric of the PBD, CBD, and a non-iterative algorithm that selects all pivot bits at once [14, 15]. The input is a synthetic rule set. ........................................ 151

8.7 Quality of partitioning of the PBD and CBD using a benchmark rule set . 151

8.8 Evaluation of 1-GREEDY, 2-GREEDY, and 3-GREEDY, for various values of \( n_s \), \( f \), and \( P_n \). ........................................ 152

8.9 Fraction of the optimal valid coloring size that can be achieved by the 1-
GREEDY, 2-GREEDY, and 3-GREEDY algorithms. Parameters are \( n_s = 7 \), \( f = 7 \) and \( P_n = \frac{5}{7} \). ........................................ 153

8.10 Evaluation of 1-GREEDY, 2-GREEDY, and 3-GREEDY for the multicolored switch case, with various values of \( n_s \), \( f \), and \( P_n \). ........................................ 154
Abstract

Tables that match a given key to its value (e.g., hash tables) have become crucial algorithmic building blocks for contemporary networking devices, which typically handle large amounts of data at high speeds. For example, these tables are used for heavy-hitter flow identification, flow state keeping, virus signature scanning, flow counter management, IP address lookup algorithms, and access control. Networking devices tables require a strict worst-case operation time along with limited table size, unlike traditional table-based data structures where amortized operation time is typically the main figure of merit.

In this dissertation, we first consider multiple-choice hashing schemes, which are particularly suited to such a worst-case operation time bound. Assuming a uniform memory model and an overflow list, often implemented using Content-Addressable Memory (CAM), we start by considering multiple-choice hashing schemes without deletion operations, i.e. only insertion and query operations. In this case, we find optimal online hashing schemes with guaranteed worst case operation time.

Then, we extend the model by considering also cases where deleting elements is allowed. While previous results show the same asymptotic behavior between the two models, we show that when bucket sizes are bounded the performance may dramatically decrease. In particular, in the deletions case, optimal online schemes decrease the expected overflow fraction as slowly as $\Omega(1/a)$, compared to $\Omega(e^{-a})$ in case deletion are not allowed, where $a$ is the average number of memory accesses. We also study a balancing problem variant, where elements need to be balanced among the buckets using only few memory accesses. The construction of an energy-efficient Bloom filter-like data structure is a direct application of this problem.

In addition, we consider a combined memory model (e.g., of both SRAM and DRAM), in which some of the elements are stored in a fast memory, while others are stored in a significantly slower memory. In this case, we provide a tight upper bound on the number of elements that can be stored in the fast memory when using a multiple-choice hashing scheme with $d = 2$ choices.

Finally, we show how network rules usually implemented using Ternary Content-Addressable Memories (TCAMs) at the ingress points of the network can be split among all routers in the network, such that each switch can maintain a smaller TCAM, while preserving the same overall classification functionality.
Abbreviations and Notations

\( n \) — Number of elements
\( m \) — Number of buckets
\( h \) — Bucket size
\( H_i \) — Hash function with index \( i \)
\( T_i \) — Subtable with index \( i \)
\( \alpha_i \) — Proportion of subtable \( T_i \)
\( d \) — Worst case number of memory accesses
\( a \) — Average number of memory accesses / Average number of choices
\( c \) — Load
\( \gamma \) — Overflow fraction
\( \alpha \) — Insertion probability (Chapter 5)
\( \bar{h} \) — Bucket load threshold for inserting elements (Chapter 6)
\( k \) — Number of hash functions in Bloom filter
\( B \) — Memory block size in bits
\( \beta \) — Bits-per-element ratio
\( \phi \) — Cost Function
\( \mu \) — Occupancy (elements) (Chapter 7)
\( \psi \) — Occupancy fraction
\( n_s \) — Number of switches in the network (Chapter 8)
\( m_l \) — Number of links in the network
\( w \) — Number of subtables / Number of colors
\( \lambda \) — Color assignment function
CAM — Content Addressable Memory
TCAM — Ternary Content Addressable Memory
FPR — False Positive Rate
SDN — Software Defined Network
Chapter 1

Introduction

1.1 Network Switches and Fast Decisions

A computer network connects a collection of computers and enables communication between them using links. Due to the typical large number of computers, there cannot be a dedicated link between each pair of computers. This is similar to a town where there cannot be a distinct road between any pair of houses. To handle this issue, as cars share roads in town, each link is used to transmit data originated from and destined for multiple computers. This sharing nature necessitates the usage of junctions. The electronic device that implements a junction in a computer network is called a switch (or a router).

Like cars that are used to go from one house to another, in a packet switched computer network, the data is transmitted in units called packets. The main goal of a switch is its switching task; namely, to transmit successfully as many packets as possible arriving at its input ports to one of its output ports according to a certain forwarding rule (typically, determined by the destination of the packet), with the least possible delay. In addition, modern switches have to perform other essential tasks, which we refer to as their management tasks. These tasks usually rely on the packet header, which informs computers and switches in the network with an important meta-data, such as the destination and origin of the packet.

In some cases, the route and resources allocated to a packet are determined not only by its destination address, but also by other header fields (typically, a five tuple of the source and destination addresses, source and destination ports, and transport protocol). The task of comparing the packet header with a multi-field rule set is called packet classification. It is essential mostly for security and quality of service (QoS) purposes. Other functions that require packet classification include network address translation (NAT), metering, traffic shaping, policing, and monitoring.

Modern switches are also required to keep many statistical pieces of information. These are later used for traffic engineering, admission control, and billing. For example, an important task is to find heavy-hitter flows; that is, flows that consume more than some fixed
fraction of the link’s bandwidth. Additional interesting pieces of information include the flow size distributions and the number of flows per input and output port.

In many of the above management tasks, the switch often uses tables. For example, in the access-control task, which is a special case of packet classification, upon arrival of a new packet, the switch must determine whether this packet is to be admitted or dropped, often by examining some fields in the packet header. This is usually performed using a lookup operation in a table that maintains the security policy data. Since the widely-used IPv4 protocol has up to $2^{104}$ possible values in the five relevant header fields, it is infeasible to perform the access-control task with a direct-access table. Thus, a more sophisticated table mechanism is needed.

One can take advantage of the fact that in many cases addresses are allocated contiguously (e.g., in an IP sub-network, where all IP addresses share the same prefix); thus, the switch can store only one entry to represent the entire network instead of an entry per computer. Hence, it can be implemented using a table mechanism relying on a Ternary Content-Addressable Memory (TCAM), that can match multiple input strings to one table entry.

Other management tasks, such as measurement tasks, may also simply rely on hash tables, which use memory that is proportional to the number of actual packet header instances that need to be stored. In the next section we describe these tables.

1.2 Tables in Switches

1.2.1 Hash Table

The hash table is a data structure that efficiently maps certain elements to associated values. It uses hash functions to transform an element into its hash value, i.e., the index in an array of buckets where the corresponding value is to be sought. For a given element and its associated value, the hash table supports an insertion operation by inserting the element-value pair into the corresponding bucket, such that a later query operation on this element will return its associated value; a query operation on an element that is not in the hash table returns a special value, indicating that the element is not in the table. Likewise, some hash tables also support a deletion operation. For an easier discussion, we assume in the sequel that only elements are stored in the hash table (without any value); thus, we define the hash-table functionality as maintaining a set with insertions, deletions, and query operations.

Ideally, the hash function should map each possible element to a different bucket; but this is rarely achievable in practice. Thus, hash table designs assume that hash collisions—different elements with the same hash values—occur occasionally, and accommodate them in some way. Traditional strategies, such as open addressing [16] and chaining [17], handle the memory dynamically or store the element in another bucket, resulting in relatively costly algorithms for insertion, deletion, and queries. Thus, such traditional implementations
cannot ensure a constant run time in the worst case upon a query decision (e.g., a decision needs to be taken), making them poorly suited to high-speed networking devices [18]. Further, when insertion times become unacceptable, the traditional solution of performing a full rehash, where all elements are rehashed using new hash functions, is also impractical.

1.2.2 Content-Addressable Memory (CAM) and Ternary Content-Addressable Memory (TCAM)

The Content-Addressable Memory (CAM) is a fully associative memory that can store a set of fixed-length binary strings. It supports a lookup of a binary string (e.g., certain bits in a packet header) in a single memory access by performing lookups on all memory locations in parallel, and returning the index of the matching string (if any).

To support string aggregation, a Ternary Content-Addressable Memory (TCAM) chip stores an ordered set of rules that consist not only of the binary bits ‘0’ and ‘1’, but also of a special value ‘*’ (don’t care value).

Since TCAMs are usually used for taking decisions in a switch (e.g., whether the packet is to be admitted or dropped), a data reflecting the decision need to be taken is stored along with each rule.

As in the CAM, the basic operation of the TCAM is to match a given binary string to the rules. Given a binary string, the TCAM chip compares it in parallel to all of the rules and returns the data associated to the first rule entry that matches the binary string. This implies that the order of TCAM entries determines their priority.

The parallelism of both the CAM and TCAM chips makes them extremely power-hungry and therefore its size is limited. Some implementations of OpenFlow, for example, limit the number of rules in each such table to only 750 [19], while today’s commodity switches support just $2K$ to $20K$ rules [20, 21].

1.3 Specificity of Networking Devices

Hash tables implemented in networking devices (e.g., switches) lay on their critical path, and thus, must guarantee a worst-case operation time.

A typical approach is to restrict the data structure such that the worst-case number of memory accesses per element operation is constant. For example, we may restrict the worst-case number of memory accesses upon a query operation, and therefore, some decision can be made within worst-case operation time. Furthermore, in some cases it is also needed to restrict insertions and deletions (e.g., upon an update). Multiple-choice hashing schemes are particularly suitable to guarantee a worst-case query time of $d$ [22, 23]. This is because in these schemes, each element can only reside in one of $d$ possible fixed-size buckets. For instance, in the $d$-random and $d$-left schemes [24, 25, 26], each arriving element uses $d$ hash functions to check the states of $d$ buckets, and then joins the least-occupied one. Then, to query if an element is in the table, it suffices to query the corresponding $d$ buckets.
In such an implementations, the bucket size is fixed, and in order to achieve a worst-case operation time, an additional simple data structure is used to store elements that cannot fit into a bucket; we refer to such elements as overflow elements. The proportion of overflow elements out of all elements is called the overflow fraction.

1.4 Performance Metrics

Hash-based algorithms and data structures are commonly measured by their space requirements, the speed of various operations, and their energy consumption. In this section, we explain these measures when a combination of SRAMs, DRAMs, CAMs, and TCAMs forms the memory.

1.4.1 Speed

In conventional algorithm analysis, speed is measured in terms of processing steps (e.g., instructions). However, in hardware, multiple complex processing steps can be performed in a single cycle using combinatorial logic gates. Therefore, only the number of serial memory accesses to the slowest memory is usually accounted for.

The memory used in hardware is usually based on SRAM or DRAM technology. Typically, on-chip SRAM access times are 1-2 ns, and off-chip SRAM access times are 5-10 ns. But, on-chip SRAMs are limited to approximately 64 Mbits today [27], which means that the fast memory cannot have a large size. On the other hand, it takes 40-60 ns to access the fastest off-chip DRAM, with an even longer worst-case time needed between successive reads (e.g., 100 ns). This implies that high-capacity DRAMs are at least 10 times slower than SRAMs.

Therefore, when the memory used by the scheme is not uniform, a common measure is to count only the number of serial memory accesses to the slowest memory. When the
memory is uniform, the speed is determined by the number of serial memory accesses (to the main memory).

1.4.2 Space Requirement

The main space measure is typically the amount of fast memory required for an algorithm or a data structure: An SRAM bit cell requires at least 5 transistors, while a DRAM cell uses only a single transistor connected to an output capacitance that usually takes much less space than the transistors in an SRAM. Thus, SRAM is less dense and more expensive (per bit) than DRAM.

Switches often make use of CAMs and TCAMs. Both are based on SRAM technology and has additional circuits thus. Furthermore, in practice, their size is also limited. Thus, when it exists, their size is usually the main space metric.

Figure 1.1 shows a setting of a hash table with a simplistic uniform off-chip memory model. Traffic that enters the chip need to be queried in the hash table, whose memory resides on an off-chip memory (i.e., DRAM). Since each query may require $d$ memory accesses, this means that the links to the DRAM need to work with a speedup of $d$; equivalently, given a total chip in/out pin capacity, this implies that a larger fraction of this capacity is devoted to hashing, yielding a significant throughput decrease as $d$ increases. Hence, as illustrated in Figure 1.1, the large average number of memory accesses into off-chip DRAM memories translates into a lower throughput.

1.4.3 Energy Considerations

Energy is measured by the total number of memory accesses and the total number of hash computations. Unlike the speed measure, where only the number of serial memory accesses counts, energy consumption does not depend on the parallelism of the hashing scheme.

The parallel lookup nature of the CAM and TCAM causes them to consume significantly more power than comparable SRAMs, while their size is proportional to their energy consumption. Thus, if the switch uses a CAM or a TCAM, then the number of memory accesses to them is the significant performance number also in terms of energy consumption.

1.5 Dissertation Overview and Main Contributions

As detailed below, in this dissertation, we first consider hashing schemes with a uniform memory model and an overflow list. We start by considering multiple-choice hashing schemes without deletion operations, i.e. only insertion and query operations. Then, we overview our results in cases where deletions are also allowed. Next, we study a balancing problem variant, where elements need to be balanced among the buckets using only few memory accesses. Construction of an energy-efficient Bloom filter-like data structure is a direct application of this problem.
This is followed by considering a combined memory model (e.g., of both SRAM and DRAM), in which some of the elements are stored in a fast memory, while others are stored in much slower memory.

Finally, as mentioned before, some network tables store network rules (for example, forwarding or access control rules). Those network rules are usually implemented in the ingress points of the network, and typically using TCAMs. We show how these tables can be split to smaller tables and be stored in other switches, while their semantic is preserved.

1.5.1 The Static Case

We first consider the case where only element insertions (and queries) are allowed (i.e., no deletions), and, by relaxing the constraint on $d$, we establish a lower bound on the expected overflow fraction as a function of the average number of memory accesses $a$ per element insertion. This lower bound also depends on other system parameters such as the bucket size and the load of the hash table (that is, the ratio between the number of elements and the total memory size).

The lower bound enables us to prove the optimality of three online hashing schemes that we propose. We show that each of them is optimal for a specific range of values of $a$.

While we focus on insertions, many of our results also apply to query operations (a.k.a. lookups) of elements in the hash table. In particular, in all our proposed schemes, the expected number of memory accesses per element lookup in the hash table is $a$, as it is for element insertion. It is important to notice that, while for obtaining the lower bound we relax the constraint on $d$, our optimal hashing schemes conform with this constraint; namely, the worst-case number of memory access for either element insertion or element lookup is $d$. Therefore, a chip designer who wants to use these schemes can fully characterize both their lookup and insertion performances.

1.5.2 The Dynamic Case

We next turn to the dynamic case, in which element deletions are also allowed. We call the (infinitely long) sequence of these operations the input sequence of the scheme. The input sequence greatly affects the performance of the hashing scheme. Therefore, we focus mostly on specific type of input sequences that alternate between a single deletion of a random element (picked uniformly at random) and a single insertion of a new element [24, 28].

We first consider a simplistic dynamic scheme with a single hash function. We model this hashing scheme analytically using two different models: a discrete-time Markov chain, and a fluid model with a continuous-time Markov process. We find that this simplistic dynamic scheme performs notably worse than its corresponding static scheme.

Then, we obtain a lower bound on the expected overflow fraction in the dynamic model of any hash-table scheme that uses uniform hash functions. We prove that when the average number of memory accesses per insertion $a$ increases, the expected overflow fraction can decrease as slowly as $\Omega(1/a)$ (compared to $\Omega(e^{-a})$ in the static case). This indicates that the
poor performance of dynamic schemes is inherent, and cannot be mitigated by simply using additional memory accesses (or hash functions). We match the lower bound by introducing an online multiple-choice scheme that is optimal up to a certain rate of memory access, which depends on the system parameters.

1.5.3 The Balancing Problem and Its Application to Bloom Filters

We next consider a Bloom filter, which is a space-efficient randomized data structure that supports approximate set membership queries. Its accuracy is measured by its false positive rate (FPR), i.e. the probability that a set membership query returns true, while the element is not in the set; Bloom Filters always have zero false negative rate. The Bloom filter can be used as the underlying data structure for various management tasks in packet switches, such as address lookup and virus signature scanning.

While Bloom filters are efficient in memory space, they require a significant number of memory accesses. For instance, a Bloom filter with 30 bits per element yields negligible FPR, but also requires to access about $30 \cdot \ln 2 \approx 21$ memory bits per query. Each of these bits can reside in an arbitrary location over the memory space. Thus, such a Bloom filter would need a prohibitive memory access bandwidth in networking devices when either implemented in an off-chip setting (that is, requiring to access memory blocks per query) or distributed over a network (equivalently, it may be required to access 21 nodes per query).

We want to build a Bloom filter-like data structure that is both efficient in memory space and memory bandwidth. A promising approach is to spread the elements between memory blocks and represent each subset of the elements as a local Bloom filter (residing in one specific memory block) [29, 30, 31]. The spreading of elements to block is done using hash functions. Thus, the problem then turns into balancing the elements between the memory blocks, with the constraint that each element can reside only in up to $d$ memory blocks.

We study this problem using a general access-efficient approach to the balancing problem. Following the same uniform memory model and an overflow list, the quality of the balancing is measured by the load on the memory blocks: The resulting load at each block induces a certain cost, which is calculated by an arbitrary non-decreasing convex cost function $\phi$. Our goal is to minimize the overall expected cost of the system, given the restriction that each operation can look at up to $a < d$ blocks (“bins”) on average, before deciding where to place the element.

As before, we first provide lower bounds on the minimum cost of each instance of the problem. The lower bound depends on the access budget $a$, the number of hash functions $d$, and the overflow list size, but, quite surprisingly, does not depend on the cost function $\phi$. Our lower bounds hold when all hash functions have uniform distribution or when their overall distribution is uniform (in the latter case, the hash function distributions can be different). Then, we provide three different schemes that meet the lower bounds on different access budgets; we further find the minimum size of the overflow list that should be provided.
in order to achieve optimality. All our analytical models are compared with simulations showing their accuracy.

We conclude by showing how, with a proper choice of the cost function $\phi$, the balancing problem can be directly used to optimize Bloom filters. For example, for an average number of access operations of $a = 1.2$, and $\gamma = 0.5\%$ of the elements stored in the overflow list, the FPR is reduced by up to two orders of magnitude.

1.5.4 Combined Memory Model via General Lower Bound

We also consider a combined memory model, in which some of the elements are stored in a fast memory, while others are stored in much slower memory. Assuming that multiple-choice hashing scheme is used to maintain the fast memory, and overflow elements are stored in the slow memory, we study the tradeoff between the load of a hash table and its average lookup time. The problem is solved by analyzing an equivalent problem: Finding a lower bound on the number of elements that need to be stored in the overflow list.

Our basic approach is to consider the expected maximum matching size in a random bipartite graph with left-vertex degree 2 that corresponds to any multiple choice hashing scheme with $d = 2$. We decompose each random bipartite graph into connected components, and then separately analyze each component and evaluate the size of its local maximum bipartite match. The size of the maximum bipartite matching is the sum of the sizes of all local matches. Then, we count the number of connected components in the graph and thus derive the size of the maximum matching in the entire graph. Surprisingly, we can obtain an exact expression of the average performance of cuckoo hashing in any finite system.

Then, since the maximum matching size of the graph is always an upper bound on the number of elements that can be inserted into the hash table, we obtain a lower bound on the number of elements that need to be stored in the overflow list, regardless of the hashing scheme that is used.

1.5.5 Spreading Tables Across The Network

Finally, we want to cope with the typical limited size of available memory (and in particular, TCAM). To do so, we introduce the Palette framework for distributing tables, whose elements represent network rules, into a network of heterogeneous switches with tables of limited size. We assume that rules are stored in an aggregated form, e.g. as in TCAM devices. Furthermore, each rule has an associated action that should be taken upon a match. We further assume that the network has a controller with a global view, and therefore, knows the paths taken by all packets. This is for example common in Software Defined Networks (SDNs).

To split the rules across the network, such that each switch will have a smaller SDN table, we divide the problem into two separate subproblems.

The first subproblem is to decompose a large table that contains all rules into a predetermined number of smaller tables, so that if all of the subtables are queried, in any order,
the resulting action will be the same as if the original large table was queried. Incidentally, this subproblem is also useful in other contexts, such as achieving parallelism and power-efficiency in TCAMs [14, 15].

After obtaining a set of small tables, the second subproblem is to ensure that each packet that goes through the large table, will traverse all small substables, so that the resulting setting would be semantically equivalent to a single lookup in the large table.

We model this second problem in a graph-theoretic manner, in which the switches are nodes and the links are edges. We assume that all the possible end-to-end paths are known to the controller. Thus, our goal is to assign colors to the nodes such that each path is a rainbow path—a path whose nodes include all the colors. The objective is to maximize the number of colors that we can use, and therefore, to minimize the table size residing at each node.

Although constructing an optimal assignment is NP-hard, we show that when the network is a tree (which corresponds to datacenters) the problem is tractable. We then propose and evaluate sub-optimal greedy algorithms, and show that they achieve close-to-optimal results in practice.

Finally, we also study the the case where each node can have more than one type of sub-table. The additional degree of freedom enhances the flexibility of our Palette distribution framework.
Chapter 2

Related Work

In this chapter we survey the relevant work to our research. In Section 2.1 we present the relevant background on multiple-choice hash tables and their variants. Section 2.2 deals with related work on Bloom filters. Next, we survey the most relevant work on load-balancing in Section 2.3, and on matching problems in graphs in Section 2.4. Last, Section 2.5 describes prior work on network coordination and software defined networks.

2.1 Multiple-Choice Hash Tables

Unlike traditional hash-table schemes, hash tables that are used in packet switches must guarantee for a worst-case operation time. Providing such a guarantee is hard due to the collisions between elements. Traditional strategies include open addressing and chaining [16]. However, these strategies only provide algorithms for insertion, deletion, and lookups that do not ensure a constant run-time in the worst case.

A popular approach to achieve worst-case operation time with a reasonable overflow fraction is to use multiple choice hash tables.

2.1.1 $d$-random and $d$-left

In the $d$-random hashing scheme, which was introduced in [24], each element uses $d$ hash functions to obtain up to $d$ possible buckets, and inserted in the least occupied one; ties are broken randomly. If one inserts $n$ elements sequentially to $n$ infinitely-sized buckets, the maximum load in a bucket (with high probability) is $\frac{\log \log n}{\log d} + \Theta(1)$, compared with maximum load of $\frac{\log n(1+O(1))}{\log \log n}$ in traditional schemes with a single hash function.

An improved scheme, called $d$-left, was introduced in [25, 26]: Initially the $n$ buckets are divided into $d$ sets of size $\frac{n}{d}$ each, which are ordered from left to right. Each element is hashed by all $d$ hash functions, each with range of $[1... \frac{n}{d}]$. The $i$-th hash function determines the index of the bucket within the $i$-th set of buckets. The element is placed in the least occupied bucket, where ties are broken to the left. This scheme was shown to achieve a maximum load of $\frac{\log \log n}{d \log d} + \Theta(1)$, where $\phi_d$ represents the exponent of growth for a
generalized Fibonacci sequence ($\phi_2 = 1.61 < \phi_3 < \phi_4 < \ldots < 2$) [25]. Counter-intuitively, it performs better than d-random.

2.1.2 Multi-Level Hash Table

Unfortunately, both d-random and d-left schemes require d memory accesses for every query, insertion and deletion. Although this can be done in parallel, it is still inefficient when considering power consumption. This problem is addressed by the multi-level hash table (MHT) in [32, 33, 34]. MHT divides the hash table into d ordered sub-tables. One hash function is used for every sub-table, but unlike the d-left scheme, each element is placed in the first sub-table that is not fully occupied. Thus, it is most likely that less than d memory accesses are used for an insertion operation.

The specific partition of the memory into subtables significantly affects the performance of the MHT. Usually, by a heuristic rule-of-thumb, subtable sizes decrease geometrically. In such a case, a prior work on MHTs [32] considers implementations where one must be able to insert n elements into some MHT with $O(n)$ finite-sized buckets and be assured that, with high probability, all elements are inserted into the buckets (with no overflow list). It was shown this can be done with an expected constant number of re-hashings of the elements for $d = \log \log n + O(1)$.

2.1.3 Hash Table Schemes With Moves

Taking advantage of the fact that every element has d possible buckets, cuckoo hashing was introduced to improve space utilization [35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45] Any operation follows exactly the d-random scheme. However, if an insertion operation fails because all buckets are occupied, the new element switches with an older element, which is rehashed and may switch in turn with another element, and so on. This process continues until all elements are placed in non-full buckets.

In case of a bucket size 1 and a load below 50%, insertions succeed in expected constant time, even when considering re-hashings [46]. Yet, a single insertion operation may require $\Omega(\log n)$ moves with non-negligible probability.

Ever since, the main effort has been to find a load threshold, such that for any load below the threshold, a perfect matching exists with high probability. It is known that a cuckoo hashing scheme with $d = 2$ succeeds to store all elements with high probability if the load is less than 0.5, but fails when the load is larger [35]. Recent works [37, 38, 39] have also settled the problem of finding the corresponding thresholds for $d > 2$. Moreover, [42] shows that cuckoo hashing with a stash of size $s$, $d = 2$, and a load less than 0.5 fails with probability $O(n^{-s})$.

The multiple moves nature of cuckoo hashing makes it hard to implement the cuckoo hashing scheme in hardware, since worst-case operation time is prohibitively high. Moreover, this scheme is highly energy consuming. To resolve these problems, a variation
of the MHT scheme, where for every insertion only one element move is allowed, was pro-
posed in [47]. It was shown that such a scheme significantly increases the space utilization
compared to the traditional MHT.

2.1.4 Hash Table With Deletions

Hash tables with deletions, which we refer to as dynamic hash tables, are known for being
typically harder to model than static ones, sometimes even lacking any mathematical anal-
ysis [22]. Therefore, the static model (i.e., no deletions) appears to be a simpler and more
accessible option.

More significantly, former studies have found the same asymptotic behavior in dynamic
and in static hash tables, in at least three cases. These studies considered both the static
case in which $n$ elements are only inserted into $n$ buckets of infinite size, and a specific
dynamic model, in which a fixed load of $n$ elements is kept by alternating between a
randomly chosen element deletion and a new element insertion:

(a) In case the elements are uniformly hashed into the buckets, the maximum bucket size
is known to be approximately $\log n / \log \log n$ with high probability. The dynamic model
yields the same result [48, 23].

(b) Likewise, when inserting each element in the least-loaded of two random buckets
($d$-random algorithm with $d = 2$), the maximum bucket size is $\log \log n / \log 2 + O(1)$ in
the static case; and again, the dynamic case yields the same result [23, 24].

(c) Similarly, using the asymmetric $d$-left algorithm [25], the static case and the dynamic
case yield again the same bound on the maximum bucket size [26].

However, it appears that in systems with finite-size buckets the behavior is not the same.
In the literature, several solutions exist to reduce the drop rate (or collision probability)
in a dynamic system. One such solution uses limited hash functions in order to be able
to rebalance the hash table in case of deletion [18]. However, this approach gives up
randomness, and the efficiency of a similar approach appears limited [28]. Another solution,
based on the second-chance scheme [47], moves elements from one bucket to another by
storing hints at each bucket [28]. These hints help to find another element stored in another
bucket that can be moved upon the deletion.

2.1.5 Hash Table With a Combined Memory

Additional papers consider the problem of off-chip memory. When the SRAM is too small,
an on-chip summary of the off-chip elements is used to reduce the average number of off-chip
accesses to almost 1 per element query [34]. But note that as a consequence, an off-chip
access is performed in any hashing operation. Another non-uniform memory model-based
hashing scheme is the peacock hashing, which also stores clues in on-chip memory and
improves upon MHT for deletions [18]. Other related aspects of multiple-choice hashing in
multiple disks is also found in [49].
2.2 Bloom Filter And Its Variants

A Bloom filter [50] is a space-efficient randomized data structure that supports set membership queries. It can be used as the underlying data structure for various management tasks in packet switches, such as address lookup and virus signature scanning. We start by reviewing the traditional concept; then, we describe in details some high-speed and energy efficient variants. Examples of well-known products using Bloom filters and their variants are Mellanox’s IB Switch System [51], Google’s database system BigTable [52], and the Web Proxy Cache Squid [53].

2.2.1 Bloom Filter

A Bloom filter for representing a set \( S = \{x_1, x_2, \ldots, x_n\} \) of \( n \) items from a large universe \( U \) consists of an array of \( m_b \) bits, initially all set to 0. The filter uses \( k \) independent (fully random) hash functions \( h_1, \ldots, h_k \) with range \( \{1, \ldots, m_b\} \). For each item \( x \in S \), the bits \( h_i(x) \) are set to 1 for \( 1 \leq i \leq k \). To check if an item \( y \) is in \( S \), all \( h_i(y) \) are checked whether they are set to 1. If not, then clearly \( y \) is not a member of \( S \). If all \( h_i(y) \) are set to 1, \( y \) is declared to be in \( S \). Hence, a Bloom filter may yield a false positive which happens when \( y \notin S \) but all the corresponding bits were set by other elements in \( S \).

Figure 2.1 illustrates the Bloom filter. The filter begins as an array of all 0s. Each item in the set uses \( k = 3 \) hash functions, with each hash yielding a bit location; these bits are set to 1. To check if an element \( y \) is in the set, apply the \( k \) hash functions on \( y \) and check the corresponding bits. The element \( y_1 \) cannot be in the set, since a 0 is found at one of the bits. The element \( y_2 \) is either in the set or the filter has yielded a false positive.

The false positive probability, a.k.a. false positive rate (FPR), can be estimated in a straightforward fashion, given that the hash functions are fully random. After all the items of \( S \) are hashed into the Bloom filter, the probability that a specific bit is still 0 is
\[ p' = \left(1 - \frac{1}{m_b}\right)^{kn} \approx e^{-\frac{kn}{m_b}}. \] Thus, the false positive probability is \( \left(1 - e^{-\frac{kn}{m_b}}\right)^k \), namely, the probability to encounter \( k \) 1-bits. This expression is minimized when \( k = \ln 2 \cdot \left(\frac{m_b}{n}\right) \), giving a false positive probability \( \left(\frac{1}{2}\right)^k \approx (0.6185)^{m_b/n} \). In practice, \( k \) must be an integer, and a smaller, sub-optimal \( k \) might be preferred since this reduces the number of hash functions that must be computed.

Occasionally, variants of the standard Bloom filter are compared by the false positive probability for a given bits-per-element ratio, i.e., the ratio of the total number of bits \( m_b \) by the number of elements \( n \).

### 2.2.2 Blocked Bloom Filter

Standard Bloom filters are cache-inefficient since up to \( k \) cache misses may be generated by each operation. In [29], a cache-efficient variant called blocked Bloom filter was introduced. It consists of a sequence of \( b \) comparatively small standard Bloom filters, that is, Bloom filter blocks, each of which fits into one cache-line or large memory-word\(^1\). As illustrated in Figure 2.2, for each element, the first hash value selects the Bloom filter block to be used. Additional hash values are then used to set or test bits as usual, but only within this single block. A blocked Bloom filter therefore has at most one cache miss per operation and, in case of large memory-words, it requires only a single memory access.

On the other hand blocked Bloom filters have higher false positive rate than the standard Bloom filters. The difference in the performance between the two is larger as the bits-per-element ratio increases, while for low bits-per-element ratios (e.g., 8) the performance is almost the same.

\(^1\)Nowadays, a common cache line size is 512 bits [29].
2.2.3 Bloom Filters via $d$-left Hashing and Dynamic Bit Reassignment

Another scheme with the same functionality as the Bloom filter is the $d$-left Bloom filter, which is based on $d$-left hashing and was introduced in [31]. The scheme relies on the fact that if $n$ elements are hashed according to $d$-left scheme to a hash table with $n$ buckets divided into three subtables each with $\frac{n}{3}$ buckets then the probability that in a certain bucket there are 6 or more elements is bounded by $10^{-30}$ [54, 25].

The memory is divided as described and upon insertion an element is given a fingerprint, i.e. a short bit-string representation of the original element, which is stored in the corresponding bucket. The fingerprint size is such that it will share the whole bucket memory equally with the other fingerprints that are already stored in the bucket. Consequentially, upon insertion of a new fingerprint to the bucket, the other fingerprints in the buckets are cut to the appropriate size. This technique is referred to as dynamic bit reassignment and is used as well as other techniques to improve the false positive rate of the standard Bloom filter.

2.2.4 Energy-Efficient Bloom Filters

In a standard Bloom filter, for any query operation, each one of the $k$ hash functions is queried and as a result, there are $k$ memory accesses. Since some of the bits may be set to 0, there is no need to read the subsequent bits. Thus, it is possible to divide the computation of the hash functions into $K > 1$ stages and query one stage at a time. This construction is called a pipelined Bloom filter [55, 56, 57].

In a two-stage pipelined Bloom filter, the $k$ hash functions are divided among the two stages. The less hash functions are implemented in the first stage of a pipelined Bloom filter, the less power is consumed in the first stage, but then there might be more accesses to the second stage, hence consuming more power there and incurring a clear tradeoff.

Thus, the challenge in the two-stage pipelined Bloom filter is to determine the number of hashing functions in each stage. Also, for each queried element, the configuration (number of hashing functions) in each stage can be either variable or constant. In the first case, some configuration overheads is needed and in the second case the configuration does not operate efficiently.

2.3 Load Balancing

The balancing problem has been extensively investigated in the last decades for various applications involving allocations of resources [58]. Since this is hardly the scope of this dissertation, we only briefly review the work related to the balancing problem.

Prime examples of the balancing problem include task balancing between many machines [59], item distribution over several locations [60], bandwidth allocation in communication channels [58] or within switches and routers [61], and hash-based data structures [48].
2.4 Random Graph Matching Problem

As in the work surveyed in 2.1.3 (e.g., [37, 38, 39]), the work in [62, 63] also provides the probability of a perfect matching but in other random bipartite graph models. However, it also does not provide the expected maximum matching size when this probability is different from one.

The expected maximum matching size is investigated in [64, 65] for an \((r,s)\)-tree, i.e., an acyclic bipartite graph with \(r\) left vertices and \(s\) right vertices. The authors show that the expected maximum matching size of almost all \((n,n)\)-trees is at least 72% of the number \(n\). A more recent work [66] presents results related to the expected maximum matching size of the class of other simply generated trees.

The problem of finding the expected maximum matching size has been also investigated assuming other models of random graphs. A model of a loop graph was considered by [67], showing a lower bound on the expected maximum matching size. While using the cavity method of statistical physics [68], the authors found analytically the value under consideration for Erdős graph \(G(n,c/(n−1))\), where \(c < 2.7183\).

2.5 Software Defined Networks and Network Coordination

Splitting the workload between the switches in a coordinated network has also been proposed in the past in many contexts. Such contexts includes traffic engineering [69], network diagnosis [70, 71], intrusion detection [72], and traffic monitoring [73, 74, 75, 76, 77, 78, 79]. However, these solutions are not directly applicable to our case. For example, cSamp [75] is a generic framework for network measurement, where each flow is monitored only in one of the network routers. It uses a hash function with a certain distribution at each router to determine whether the current router has to perform the measurement. Since the hash functions are orthogonal to the monitoring rules, it implies that each such router should hold the entire monitoring table (but only access the table on a subset of the packets).

Another approach for distributing table rules across the network is DIFANE [80] (not necessarily in SDNs). In DIFANE, non-overlapping flow ranges are allocated using a decision-tree based algorithm. Then, each such range is assigned to a different predetermined subset of the switches, called authority switches. Following that, rules are installed to these authority switches according to the corresponding flow range.

To assure that each packet is matched against all relevant rules, ingress switches redirect the packets to the corresponding authority rule. If some rule is matched, a cache rule is generated using a technique described in [81], and then installed in the ingress switch, such that future packets from the same flow can be managed instantly in the ingress router.

While DIFANE does all rule processing in the data plane, a recent paper proposes to combine the rule processing both in the data plane and control plane [82]. The main approach is to partition the rules into non-overlapping sets of rules, and then to distribute it to both the switches (i.e., data plane) and the hypervisors (i.e., control plane), such
that the volume of flows need to be redirected from the data plane to the control plane is minimized.

Coordinated network mechanics, is also suitable for Software Defined Networks (SDNs). The key concept of SDNs lies in the management of the entire network as a unified abstraction (e.g., in a network controller), and the remote control of the network devices (namely, its switches and routers) through open protocols (such as OpenFlow) [83]. This property enables to maintain a coordination between the switches easily.

In recent years, SDN technology has been widely deployed in real-life large-scale networks, e.g. Google’s G-scale network [84]. Switches and routers that support SDN/OpenFlow are now offered by a large number of vendors (e.g., [19, 85, 86]).

One of the major challenges in SDN is to develop a programming language for its software development. On one hand, this programming language should be sufficiently flexible and rich to allow new network applications, but on the other hand, it should be simple and modular to reduce development and debugging times. Frenetic [87] is a prime example of such a network programming language that gives high-level abstractions to the network programmer. For example, it allows systematic updates [88] and task composition [87]. A closely-related work to ours is the extension of Frenetic to allow policy transformation of rules across the networks [89]. The authors developed a complete and sound set of axioms to allow semantically-preserving rule-rewriting in a single switch or in a chain of switches.
Chapter 3

Model Definition

In this dissertation, we mainly consider tables implemented in software in high-speed networking devices, e.g., multiple-choice hash tables. Therefore, we capture in this chapter only the common definitions and notations of multiple-choice hash tables. These are relevant for our results described in Chapters 4-7. The relevant definitions and notations for Chapter 8 regarding tables implemented in hardware (e.g., using TCAMs) are given only there.

We adopt the conventional multiple-choice hashing terminology and notations [47, 32]. Let \( B \) be a set of \( m \) buckets (or bins), where each can store up to \( h \) elements. Also, let \( E \) be a set of \( n \) elements (or balls) that should be distributed among the buckets.

We also assume that there exists an overflow list, i.e., a special bucket of bounded size \( \gamma \cdot n \) (namely, at most a fraction \( \gamma \) of the elements can be placed in the list), which can be used by the insertion algorithm at any time. For example, depending on the application, the overflow list may correspond to a dedicated memory—e.g., content-addressable memory (CAM)—in hardware-implemented hash-table, or to the loss ratio when the balancing scheme is allowed to drop elements.

In addition, the load \( c \) is defined as \( c = \frac{n}{h \cdot m} \), and we also refer to the elements-to-buckets ratio as \( r \), i.e., \( r = \frac{n}{m} \). The definition of \( r \) is used only in Chapter 6, where \( h = \infty \).

Hash tables use hash functions. In this dissertation, we assume that all hash functions are independent and we do not consider their computation time. In some cases we will make use of uniformly-distributed hash functions, as defined below.

**Definition 1** A uniformly-distributed hash function is a hash function that maps into each bucket with equal probability.

Multiple-choice hashing schemes employ up to \( d \) probability distributions over the set of buckets; these distributions are then used to generate a hash-function set \( \mathcal{H} = \{ H_1, \ldots, H_d \} \) of \( d \) independent hash functions. For each element \( x \) and each operation, the scheme can consider only the buckets \( \{ H_1(x), \ldots, H_d(x) \} \) (and the overflow list). In addition, we assume that the scheme must access a bucket to obtain any information on it.
The above definitions are illustrated in Fig. 3.1. In particular, the memory consists of \( m \) buckets, each of size \( h \), and an overflow list. There are also \( n \) elements, and each element uses up to \( d \) hash functions.

We now skip to the definition of a hashing scheme.

**Definition 2** A hashing scheme, or hash table construction scheme, consists in defining:

(i) \( d \) hash-function probability distributions over bucket set \( \mathcal{B} \), used to generate a hash-function set \( \mathcal{H} = \{H_1, \ldots, H_d\} \) of \( d \) independent random hash functions;

(ii) An insertion algorithm that sequentially inserts the \( n \) elements in the hash table. The insertion algorithm places each element \( x \in \mathcal{E} \) either in one of the \( d \) buckets \( \{H_1(x), \ldots, H_d(x)\} \) or in the overflow list. At most \( h \) elements can be placed in each bucket.

Note that our sequential insertion framework does not allow schemes that move elements after their initial insertion, such as [46, 47]. However, our lower bound on the overflow fraction in Section 4.3 does apply to these schemes as well. Furthermore, the hash functions \( \{H_1(x), \ldots, H_d(x)\} \) are given a-priori and cannot be changed. This is often the case in networking devices (specifically, when the hash functions are implemented in hardware).

We conclude this section by an example of multiple-choice hashing scheme.
Example 1 (MHT) A multi-level hash table (MHT) \cite{32, 33, 34} construction scheme conceptually divides the $m$ buckets into $d$ separate subtables, $T_1, \ldots, T_d$, where $T_i$ contains $\alpha_i \cdot m$ buckets, with $\sum_{i=1}^{d} \alpha_i = 1$.

(i) Each hash function $H_i$ is distributed uniformly over the buckets in $T_i$;

(ii) And the insertion algorithm successively places each element $x$ in the smallest $i$ such that $H_i(x)$ is not full, and in the overflow list if all such buckets are full.

Fig. 3.2 illustrates the MHT scheme with $m = 12$, $h = 1$, and $d = 2$ (the overflow list is not represented). The gray buckets are the occupied ones. Dashed arrows represent potential memory accesses that are not performed (and exist only to illustrate the mapping of the elements). We can see that element 6 is initially mapped by $H_1$ to a full bucket in the first subtable, and therefore is inserted in the second subtable, where it is mapped into an empty bucket by $H_2$. On the contrary, element 7 is directly inserted in the first subtable, and does not use $H_2$. Therefore, it only uses one memory access.
Chapter 4

Static Table Model

In this chapter, we present optimal high-throughput hashing schemes that minimize the total number of memory accesses needed to build and access an hash table. Recent schemes often promote the use of multiple-choice hashing. However, such a choice also implies a significant increase in the number of memory accesses to the hash table, which translates into higher power consumption and lower throughput for networking devices.

We propose to only use choice when needed. Given some target hash table overflow rate, we provide a lower bound on the total number of needed memory accesses. Then, we design and analyze schemes that provably achieve this lower bound over a large range of target overflow values. Further, for the multi-level hash table scheme, we prove that the optimum occurs when its subtable sizes decrease in a geometric way, thus formally confirming a heuristic rule-of-thumb.

4.1 Summary of Our Results

This chapter investigates hashing schemes with low expected overflow fraction given the worst-case and average insertion times $d$ and $a$.

We consider only the commonly-used stateless hashing schemes, in which the only way to know a bucket occupancy is to access it, and therefore allow for a distributed hashing scheme implementation. Depending on the memory model, a hashing scheme may need an extra memory access to actually write the element into the bucket. Since this extra memory access, if needed, is valid to all hashing schemes, it does not affect the competitive evaluation of the hashing schemes. Therefore, for simplicity, we do not consider this extra memory access in our analysis.

Our results apply asymptotically to hashing schemes with a large number of elements and a large memory size. Finally, we do not consider multi-level memory [90, 34] and element deletions [18]. Note that handling deletion is known for being typically harder to model, sometimes even lacking any mathematical analysis [22]. Furthermore, deletions may decrease the performance of hash schemes compared to the insertions only case [5].
We first establish a lower bound on the expected overflow fraction as a function of the average number of memory accesses \( a \). The lower bound also depends on other system parameters such as the bucket size and the load of the hash table (that is, the ratio between the number of elements and the total memory size).

The lower bound enables us to prove the optimality of the schemes we propose. We provide three hashing schemes and show that each of them is optimal for a specific range of values of \( a \).

Specifically, we first demonstrate that a simple hashing scheme that relies on a single uniformly-distributed hash function achieves an optimal overflow fraction for \( a \leq 1 \).

We further show that a multiple-choice greedy scheme with \( d \) uniformly-distributed hash functions, in which each element successively checks up to \( d \) buckets until it can be inserted, achieves optimality for \( a \leq a_{\text{greedy}} \), where \( a_{\text{greedy}} > 1 \) depends on the system parameters.

The optimality range can be further extended using a multi-level hashing (MHT) scheme \([32, 33, 34]\). In particular, among all MHT schemes, we demonstrate the optimality of those in which the subtable sizes decrease geometrically according to a factor that depends on system parameters, thus confirming a previously-known rule-of-thumb.

We then turn to analyze the greedy and the MHT schemes for the case where non-unique elements arrive, based on the analysis given for the unique elements case.

We conclude by providing simulations and quantitative results showing that our models closely reflect real-life scenarios.

While we obtain the optimal expected overflow fraction for a specific value \( a \), we can equivalently find the optimal \( a \) for a given expected overflow fraction, and potentially a corresponding optimal scheme. Thus, this chapter also provides an \textit{optimal fast hashing scheme} given a targeted overflow fraction. Wherever possible, we give a closed-form expression for the optimal average number of memory accesses \( a \) given the expected overflow fraction.

Recall that we do not consider the extra memory access possibly needed for writing the element into the bucket when an insertion occurs. Thus, many of our results also apply to lookup operations of elements in the hash table. In particular, in all our proposed schemes, the expected number of memory accesses per element in the hash table is \( a \) in the lookups, as it is in the insertions. Also, the expected number of memory accesses is at most \( d \) for any element lookup, again as it is in the insertions. Therefore, a chip designer who wants to use these schemes can fully characterize both their lookup and insertion performances.

**Chapter Organization** We start with preliminary definitions in Section 4.2. Section 4.3 provides a lower bound on the overflow fraction. Then, in Sections 4.4, 4.5, and 4.6, we present and analyze the simple, greedy, and MHT schemes, respectively. In Section 4.7, we consider the case where non-unique elements arrive. We finally evaluate all the analytical results in Section 4.8.
4.2 Problem Statement

The throughput of a hashing scheme depends on the number of memory accesses needed to store the incoming elements in the hash table. Therefore, we define a memory access time as the time needed to access a single bucket, read all of its elements, and update them. This definition corresponds for instance to an update (operation) of a single word in SRAM or DRAM memory. We assume that a hashing scheme needs to access a bucket in order to obtain any information on it; thus, if the hashing scheme tries to insert an element in a full bucket, it wastes a memory access. We also do not count accesses to the overflow list. We allow the hashing scheme to access up to \( d \) buckets before deciding which one to update. Thus, we get the following number of memory accesses for previously known schemes:

**Example 2 (d-random and d-left)** Inserting an element in the least loaded of \( d \) buckets requires \( d \) memory accesses \([24, 25, 26]\).

**Example 3 (MHT)** Inserting an element in subtable \( T_i \) requires \( i \) memory accesses, since in that case we first sequentially access \( i - 1 \) full buckets in subtables \( T_1, \ldots, T_{i-1} \), and then access the last non-full bucket in subtable \( T_i \).

We further consider two constraints. First, we require the average number of memory accesses per element insertion to be bounded by some constant \( a \geq 0 \). In addition, the worst-case number of memory accesses per element insertion is always bounded by \( d \), because an element does not need to consider any of its \( d \) hash functions more than once. Let the load \( c = \frac{n}{mh} \) denote the ratio of the number of elements to the total memory size. Then, the following definition captures these two constraints.

**Definition 3** An \( \langle a, d, c, h \rangle \) hashing scheme is a hashing scheme that inserts all elements with an average number of memory accesses per insertion of at most \( a \), and a maximum number of memory accesses per insertion of at most \( d \), when given a load \( c \) and a bucket size \( h \).

Let \( \gamma \) denote the expected overflow fraction of the elements, i.e. the expected ratio of the number of elements that cannot be stored in the buckets to the total number of elements \( n \). These unstored elements are either dropped or placed in the overflow list, which usually is more expensive than the memory buckets (e.g., when implemented in a CAM \([47]\)) or requires more memory accesses (e.g., as a linked list). Our goal is to minimize \( \gamma \):

**Definition 4** The optimal hash table construction problem is to find an \( \langle a, d, c, h \rangle \) hashing scheme that minimizes \( \gamma \) as the number of elements \( n \) goes to infinity. Whenever defined, let \( \gamma_{\text{opt}} \) denote this optimal expected limit overflow fraction.

The definitions above do not only bound the total number of memory accesses per element insertion but also per element lookup. First, since each element can only be placed
in one of $d$ buckets, the number of memory accesses needed for a lookup is bounded by $d$.
Second, in most hashing schemes, the lookup operation accesses buckets in the same order as the insertion operation. Therefore, the average number of memory accesses to query a random element in the hash table is also $a$. So, given a probability $p$ that a queried element is in the hash table, the average number of memory accesses needed for a lookup is bounded by $p \cdot a + (1 - p) \cdot d$. Therefore, in most hashing schemes, the bounds on insertion memory accesses directly translate into bounds on lookup memory accesses.

4.3 Overflow Lower Bound

4.3.1 The Cloning Method

In this section, we provide a lower bound on $\gamma_{\text{opt}}$, and therefore on the expected limit overflow fraction of any $(a, d, c, h)$ hashing scheme. We do so by relaxing three conditions.

First, we consider an offline case, in which the hashing scheme looks at all elements at once, instead of considering them in the predetermined online sequential order.

Second, we omit the bound $d$ on the worst-case number of memory accesses per element, and enable all elements to use any number of memory accesses, as long as the average number of memory accesses per element is still at most $a$. Consequentially, the lower bound we find is valid for any $d$.

Last, we dissociate the memory accesses from the elements. In other words, we hypothetically consider each memory access to a bucket as if it is made by a clone of the initial element and allow the clone to be inserted if the bucket is not full, independently of the other clones. Thus, if one element accesses two buckets, it conceptually corresponds to two clones each accessing one of these buckets, potentially corresponding to two clone insertions. The number of inserted clones after this dissociation is clearly an upper bound on the actual number of inserted elements. In our case, since $n$ elements make at most $a$ memory accesses per element on average, we will consider a set of at most $an$ clones making one memory access each and evaluate the number of such clones that are inserted.

Conceptually, the cloning relaxation is the most significant one. While it seems to provide a crude bound, we will later see that this bound is actually tight over a range of values of $a$.

Note that our lower bound also holds for schemes that allow moves, such as cuckoo hashing [46] and one-move schemes [47], since we are assuming an offline non-sequential setting.

4.3.2 Identical Hash Function Distributions

Different elements might end up using different hash functions, and therefore generate memory accesses by their clones that are distributed in a different way. We first consider the easier case in which all hash functions have the same distribution, implying that all memory accesses by the clones are distributed in the same way. Then, we later consider
the heterogeneous case, in which the hash functions do not necessarily have the same
distribution. In both cases, we eventually derive the same lower bound on the expected
limit overflow fraction.

We start with the setting in which all hash functions are distributed identically, but
the common distribution is not necessarily uniform. In this setting, the following theorem
provides a lower bound on $\gamma_{opt}$.

To prove it, we bound the expected fraction of unused memory after the insertion of
all elements. We approximate the binomial distribution of the load on each bucket by a
Poisson distribution, and supply a bound on the approximation error. Then, we prove the
theorem on the Poisson distribution, and apply the bound to conclude.

**Theorem 1** Given a load $c$, a bucket size $h$, a memory access rate $a$ and identically-
distributed hash functions, the optimal expected limit overflow fraction $\gamma_{opt}$ in the optimal
hash table construction problem is lower bounded by

$$
\gamma_{LB} (a) = 1 - \frac{1}{c} + \frac{1}{ch} e^{-ach} \sum_{k=0}^{h} (h - k) \frac{(ach)^k}{k!},
$$

and this lower bound is computed with the uniform distribution.

**Proof.** We derive the lower bound on the overflow fraction by bounding the expected
fraction of unused memory after the insertion of all elements. Let $f : \mathcal{E} \to [0,1]$ be the
distribution used by the hash functions, where $f(i)$ is the probability that a hash function
maps any element $x \in \mathcal{E}$ to a bucket $i \in \mathcal{B}$, (and $\sum f(i) = 1$).

The random variable $B_i$ that counts the number of elements mapped to a specific
bucket $i$ follows a binomial distribution with $a \cdot n$ trials of success probability $f(i)$ each.
Furthermore, since we consider the case where $m$ and $n$ are large, we approximate it by
the Poisson distribution $P_i$ with mean $\lambda_i = a \cdot n \cdot f(i)$. Let $Z$ be the random variable that
represents the ratio of the unused memory size by the total memory size. We note that $Z$
clearly depends on $f$. Then, by the linearity of expectation,

$$
\mathbb{E} (Z) = \frac{1}{mh} \sum_{i=1}^{m} \sum_{k=0}^{h} (h - k) \binom{an}{k} f(i)^k (1 - f(i))^{a \cdot n - k}.
$$

Using the Poisson approximation,

$$
\mathbb{E} (Z) \approx \frac{1}{mh} \sum_{i=1}^{m} \sum_{k=0}^{h} (h - k) \frac{(af(i))^k e^{-a \cdot f(i)}}{k!}, \tag{4.1}
$$

and we denote the right side by $\mathbb{E}_n (Z)$.

We next show that minimizing $\mathbb{E}_n (Z)$ is equivalent to minimizing $\mathbb{E} (Z)$ by bounding the
error of the Poisson approximation. Let $d (B_i, P_i)$ be the total variation distance between
the binomial and the Poisson distribution, defined by \( \sup_A \Pr (B_i \in A) - \Pr (P_i \in A) \). Barbour and Hall Theorem \[91\] yields that

\[
d (B_i, P_i) \leq \lambda_i^{-1} \cdot \left(1 - e^{-\lambda_i}\right) \sum_{j=0}^{an} (f (i))^2 \leq f (i).
\]

By \[92\], the total variation distance is equivalent to the following expression:

\[
d (B_i, P_i) = \frac{1}{2} \sum_{k \in \mathbb{N}} |\Pr (B_i = k) - \Pr (P_i = k)|.
\]

Let \( \delta_{B_i, P_i} (k) = |\Pr (B_i = k) - \Pr (P_i = k)| \), and we find the following bound on the error:

\[
|\mathbb{E} (Z) - \mathbb{E}_n (Z)| \leq \frac{1}{m_h} \sum_{i=1}^{m} \sum_{k=0}^{h} (h - k) \cdot \delta_{B_i, P_i} (k)
\]

\[
\leq \frac{1}{m_h} \sum_{i=1}^{m} h \sum_{k=0}^{h} \delta_{B_i, P_i} (k)
\]

\[
\leq \frac{1}{m_h} \sum_{i=1}^{m} h \cdot 2 f (i) = \frac{2}{m} = \frac{2 \cdot c \cdot h}{n},
\]

which implies that the error vanishes for a large enough number of buckets \( m \) (and elements \( n \)). Therefore, for such large values, we are able to minimize \( \mathbb{E} (Z) \) by minimizing \( \mathbb{E}_n (Z) \).

By substituting \( f(m) = 1 - \sum_{i=1}^{m-1} f(i) \) in the expression of \( \mathbb{E}_n (Z) \), we get:

\[
\mathbb{E}_n (Z) = \frac{1}{m_h} \sum_{i=1}^{m-1} \sum_{k=0}^{h} (h - k) \frac{(an f(i))^k e^{-an f(i)}}{k!}
\]

\[
+ \frac{1}{m_h} \sum_{k=0}^{h} (h - k) \cdot \frac{(an(1-\sum_{i=1}^{m-1} f(i)))^k e^{-an(1-\sum_{i=1}^{m-1} f(i))}}{k!}.
\]

We further denote the inner first term for each \( i \) as \( g_i (f) \) and the sum in the second term as \( g_m (f) \), so that \( \mathbb{E}_n (Z) = \frac{1}{m_h} \sum_{i=1}^{m-1} g_i (f) + \frac{1}{m_h} g_m (f) \). Note that formally \( g_i (f) \) is a function with \( m-1 \) variables: \( f(1), f(2), ..., f(m-1) \), although in practice it only depends on \( f(i) \) (unless \( i = m \)). Thus, in order to find a global minimum we will apply a second derivative test using the corresponding Hessian Matrix.

We first deal with the Hessian matrices of \( g_i (f) \) with \( i \neq m \). Since \( g_i (f) \) only depends on \( f(i) \), the Hessian matrix \( H_{g_i (f)} = [h_{ij}^i] = \left[ \frac{\partial^2 g_i (f)}{\partial f(i)^2} \right] \) of \( g_i (f) \) has only a single non-zero element:

\[
h_{ii}^i = \frac{\partial^2 g_i (f)}{\partial f(i)^2} = (a \cdot n)^2 \frac{(a \cdot n \cdot f(i))^{h-1} e^{-a n f(i)}}{(h - 1)!}.
\]

On the other hand, the Hessian matrix \( H_{g_m (f)} \) of \( g_m (f) \) (which depends on
$f(1, \ldots, f(m-1))$ has the same elements in all its entries and their value is

$$h_{jk}^m = \frac{\partial^2 g_m(f)}{\partial f(j) \partial f(k)} = (a \cdot n)^2 \frac{(a \cdot n \cdot f(m))^{h-1} e^{-a \cdot n \cdot f(m)}}{(h-1)!},$$

where $f(m) = 1 - \sum_{i=1}^{m-1} f(i)$.

It is easy to verify that all eigenvalues of these Hessian matrices $H_{g_i(f)}$ are non-negative, implying that all the functions $g_i(f)$ are convex. Thus, $E_n(Z)$ is also a convex function, as a sum of convex functions. Finally, in order to find an external point we differentiate the expression:

$$\frac{\partial E(Z)}{\partial f(i)} = -a \cdot \frac{n}{mh} \cdot e^{-a \cdot n \cdot f(i)} \cdot \sum_{k=0}^{h-1} \frac{(anf(i))^k}{k!} + a \cdot \frac{n}{mh} \cdot e^{-a \cdot n \cdot (1 - \sum_{i=1}^{m-1} f(i))} \cdot \sum_{k=0}^{h-1} \frac{(an(1 - \sum_{i=1}^{m-1} f(i)))^k}{k!}.$$

Comparing this expression to 0 yields a system of $m-1$ equations with $m-1$ variables; one of its solutions $f_u$ corresponds to the uniform distribution, in which $f(i) = \frac{1}{m}$ for all $i \in \{1, \ldots, m\}$. By applying the second derivative test on the Hessian matrix $H_{E_n(Z)} = \sum_{i=1}^{m} H_{g_i(f)}$, we verify that it is positive definite at $f_u$ and thus $f_u$ is a local minimum of $E_n(Z)$. By convexity, we deduce that it is in fact a global minimum over all the region.

Thus, the minimum expectation of unused memory is $E_{\min}(Z) = e^{-a \frac{n}{mh} \sum_{k=0}^{h} (h-k) \left(\frac{a \cdot n}{mh}\right)^k}$, implying that at least $n - (mh - mhE_{\min}(Z))$ elements are overflowed. Since we are interested in the limit expectation overflow fraction, i.e. $n, m \to \infty$, we get that the error bound in Equation (4.2) tends to zero, and therefore the result expression is tight. Finally, we obtain the claimed expression of the minimum overflow fraction of elements by substituting $c = \frac{n}{mh}$. \hfill \Box

We note that the lower bound found also holds for the *average number of choices*, rather than the average number of memory accesses. Thus, it is also applied to cuckoo hashing with $a = d$.

**Example 4** If $h = 1$, then $\gamma_{LB}(a) = 1 - \frac{1}{c} + \frac{1}{c} e^{-ac}$. Thus, for $c = 1$, i.e. $n = m$, we get $\gamma_{LB}(a) = e^{-a}$, and the lower-bound decreases exponentially as a function of the average number of memory accesses per insertion $a$. Therefore, for any constant $a$, an $\langle a, d, c, h \rangle = \langle a, d, 1, 1 \rangle$ hashing scheme can never reach a zero-overflow result. Intuitively, this can be explained by the fact that regardless of the number of used hash functions, there always exists some positive fraction of the buckets that is not mapped to by these hash functions.

This example yields the following corollary:

**Corollary 2** For $h = 1$, given a load $c$, and a target expected overflow fraction $\gamma$, the optimal average number of memory accesses $a_{opt}$ is lower bounded by

$$a_{LB}(\gamma) = \frac{-\log(1 - c + c\gamma)}{c}.$$
We note that in some cases the lower bound in Theorem 1 is less than zero making it trivial. Fig. 4.1 shows $\gamma_{lb}(a)$ for $a = 1.2$, and for different values of load $c$ and bucket size $h$. For low loads our lower bound gives values less than zero. Furthermore, the smaller the bucket size is, the larger overflow fraction our lower-bound yields.

From the practical point of view, it is also interesting to see how $a$ depends on $\gamma$. Fig. 4.2 shows $a_{lb}(\gamma)$ for $\gamma = 0.01$, that is an overflow fraction of 1%. For $h = 1$, it is plotted using Corollary 2, while for the other cases we use numerical analysis to invert $\gamma_{lb}(a)$. It is interesting to see that for a load of $c = 1$ and $h = 1$, one needs at least an average of $\approx 4.605$ memory accesses per element to achieve an overflow fraction of 1%.

### 4.3.3 Multiple Hash Function Distributions

We now consider a setting where $\ell \leq d$ different distributions over the buckets are used by the $d$ hash functions. Denote these distributions by $f_1, \ldots, f_\ell$, and assume that distribution $f_i$ is used by a fraction $k_i$ of the total memory accesses, with $\sum_{i=1}^{\ell} k_i = 1$. We now show that Theorem 1 holds also in this case.
Theorem 3 The optimal expected limit overflow fraction $\gamma_{\text{opt}}$ is lower bounded by

$$\gamma_{\text{LB}} (a) = 1 - \frac{1}{c} + \frac{1}{c h} e^{-ac} \sum_{k=0}^{h} (h - k) \frac{(ach)^k}{k!},$$

and is reached when for each bucket $i \in \{1 \cdots m\}$, $\sum_{p=1}^{\ell} k_p f_p (i) = \frac{1}{m}$, namely, the weighted average of all distributions is uniform.

Proof. As in the proof of Theorem 1, the number of elements mapped to bucket $i$ by all hash functions follows approximately a Poisson distribution with rate $\lambda_i = an \sum_{p=1}^{\ell} k_p f_p (i)$. Let $Y$ be the random variable that follows this Poisson distribution, and $X$ be the actual random variable. Then, from [91, 92], we get:

$$\sum_{k=0}^{\infty} \delta_{B_i,P_i} (k) \leq \lambda_i^{-1} \cdot (1 - e^{-\lambda_i}) \sum_{p=1}^{\ell} \sum_{k=1}^{an \cdot k_p} (f_p (i))^2 \leq \lambda_i^{-1} \sum_{p=1}^{\ell} \frac{an \cdot k_p}{\sum_{p=1}^{\ell} k_p f_p (i)} \leq \frac{\max_{p \in \{1, \cdots , \ell\}} \{f_p (i)\}}{\sum_{p=1}^{\ell} k_p f_p (i)} \leq \frac{\max_{p \in \{1, \cdots , \ell\}} \{f_p (i)\}}{\sum_{p=1}^{\ell} k_p f_p (i)}$$

Therefore,

$$|\mathbb{E} (Z) - \mathbb{E}_{\alpha} (Z)| \leq \frac{1}{mh} \sum_{i=1}^{m} \sum_{k=0}^{h} (h - k) \cdot \delta_{B_i,P_i} (k) \leq \frac{1}{mh} \sum_{i=1}^{m} h \sum_{k=0}^{h} \delta_{B_i,P_i} (k) \leq \frac{2}{m} \sum_{i=1}^{m} \max_{p \in \{1, \cdots , \ell\}} \{f_p (i)\} = \frac{2 \cdot \ell}{m}$$

Thus, the proof of Theorem 1 implies that to get global minimum over the overflow fraction, for every bucket $i$, $an \sum_{p=1}^{\ell} k_p f_p (i)$ must equal $\frac{an}{m}$. This implies that

$$\sum_{p=1}^{\ell} k_p f_p (i) = \frac{1}{m}$$

is a necessary condition for optimal overflow fraction.

Further, since any legal combination of $k_p$ and $f_p$ can be considered as a single distribution, Theorem 1 implies that a lower value of the overflow fraction cannot be found if the above condition is violated, and thus is also a sufficient condition. $\square$
Figure 4.3: An example of two different distributions (probability mass functions) \( f_1 \) and \( f_2 \) that reach the lower bound with \( k_1 = \frac{1}{3} \) and \( k_2 = \frac{2}{3} \).

\[
\begin{align*}
\text{\( f_1 \)} & = \begin{cases} 
\frac{3}{m} & i \leq \frac{m}{3} \\
0 & \text{otherwise}
\end{cases} \\
\text{\( f_2 \)} & = \begin{cases} 
\frac{3}{2m} & i > \frac{m}{3} \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

Figure 4.4: Illustration of the SIMPLE scheme.

Note that while any offline algorithm may pick its own values explicitly, we would typically like to have an online hashing scheme in which the values of \( k_p \) are picked implicitly so that \( \sum_{p=1}^{P} k_p f_p(i) = \frac{1}{m} \).

This is demonstrated in the following example:

**Example 5** In Fig. 4.3, there is an example of two distributions that under \( k_1 = \frac{1}{3} \) and \( k_2 = \frac{2}{3} \) reach the lower bound:

4.4 SIMPLE - A Single-Choice Hashing Scheme

We now want to find simple hashing schemes that can potentially achieve the overflow fraction lower bound \( \gamma_{LB} \), and therefore the optimal overflow fraction \( \gamma_{OPT} \).

To provide intuition to the reader, we start by analyzing a simplistic hashing scheme, denoted SIMPLE. This scheme only uses a single uniformly-distributed hash function \( H \). Each element is stored in bucket \( H(x) \) if it is not full, and in the overflow list otherwise.

Furthermore, to keep an average number of memory accesses per element of at most \( a \), not all elements can be inserted when \( a < 1 \). Therefore, in that case, the process stops
when a total of \( a \cdot n \) memory accesses is reached, and the remaining elements are placed in
the overflow list as well.

Fig. 4.4 illustrates **SIMPLE** with \( m = 12 \) and \( h = 1 \) (the overflow list is not represented).
We can see that element 6 is mapped by \( H \) to a full bucket, and therefore cannot be inserted.
Thus, it joins the overflow list. On the contrary, element 7 is directly inserted in an empty bucket.

By the notations in Definition 3, **SIMPLE** is an \( \langle a, 1, c, h \rangle \) hashing scheme; we will show
that it is optimal for \( a \leq 1 \).

### 4.4.1 Description by Differential Equations

In recent years, several hashing schemes have been modeled using a deterministic system
of differential equations [47, 23]. We adapt this approach in order to describe the **SIMPLE** scheme.

We start by considering that the \( j \)-th element is inserted in the hash table at time \( \frac{j}{n} \);
namely, all elements are handled by time \( t = 1 \). Furthermore, let \( F_i \left( \frac{j}{n} \right) \) denote the fraction
of buckets in the hash table that store exactly \( i \) elements at time \( \frac{j}{n} \), just before element \( j \) is
inserted, and \( \vec{F} \left( \frac{j}{n} \right) \) be the vector of all \( F_i \left( \frac{j}{n} \right) \)'s. Also, let \( \Delta F_i \left( \frac{j+1}{n} \right) \triangleq F_i \left( \frac{j+1}{n} \right) - F_i \left( \frac{j}{n} \right) \)
denote the change in the fraction of buckets that store exactly \( i \) elements between times \( \frac{j}{n} \)
and \( \frac{j+1}{n} \). Then

\[
E \left( \Delta F_i \left( \frac{j+1}{n} \right) | \vec{F} \left( \frac{j}{n} \right) \right) = \begin{cases} 
- \frac{1}{m} F_0 \left( \frac{j}{n} \right) & i = 0 \\
\frac{1}{m} F_{h-1} \left( \frac{j}{n} \right) & i = h \\
\frac{1}{m} (F_{i-1} \left( \frac{j}{n} \right) - F_i \left( \frac{j}{n} \right)) & \text{otherwise}
\end{cases}
\]

At time \( t = 0 \), \( F_i \left( 0 \right) = 1 \) if \( i = 0 \) and 0 otherwise.

The first equality shows that upon the insertion of element \( j \), the fraction of empty buckets can only decrease when the element is hashed to an empty bucket, which happens
with probability \( F_0 \left( \frac{j}{n} \right) \). Likewise, in the second equality, the fraction of full buckets only
increases when element \( j \) hits a bucket of size \( h - 1 \). Last, in the third equality, the fraction
of elements of size \( i \) either increases with probability \( F_{i-1} \left( \frac{j}{n} \right) \), or decreases with probability
\( F_i \left( \frac{j}{n} \right) \). Any such increment or decrement is by a value of \( \frac{1}{m} \).

By dividing both sides of the equation by \( \frac{1}{n} \) and considering the fact that \( n \) is large,
so that the values of \( \Delta F_i \left( \frac{j+1}{n} \right) \) are comparatively very small, we can use the fluid limit
approximation, which is often very accurate [47]:

\[
\frac{df_i(t)}{dt} = \begin{cases} 
- \frac{n}{m} f_0 \left( t \right) & i = 0 \\
\frac{n}{m} f_{h-1} \left( t \right) & i = h \\
\frac{n}{m} (f_{i-1} \left( t \right) - f_i \left( t \right)) & \text{otherwise}
\end{cases}
\]
More formally, let \( \vec{f}(t) \triangleq (f_1(t), \ldots, f_d(t)) \) be the solution of the above set of linear differential equations when assuming \( f_0(0) = 1 \) and \( f_i(0) = 0 \) for each \( i \neq 0 \). Then, by Kurtz theorems [93, 94, 95], the probability that \( \vec{f} \) deviates from \( \vec{F} \) by more than some constant \( \varepsilon \) decays exponentially as a function of \( n \) and \( \varepsilon^2 \) [47]. For further intuition behind this statement, refer to [47] and [96, Chapter 3.4].

### 4.4.2 Optimality of the SIMPLE Scheme

We solve analytically the system of differential equations to obtain the overflow fraction of the scheme and show that it is identical to the lower bound given in Theorem 1. Since SIMPLE does not perform more than one memory access per operation, this yields the following theorem.

**Theorem 4** The SIMPLE scheme solves the optimal hash table construction problem for \( a \leq 1, \ d = 1 \), and any values of \( c \) and \( h \).

**Proof.** We solve the differential equations one by one, substituting the result of equation \( i \) into equation \( i + 1 \). The first equation depends only on \( f_0(t) \), thus \( f_0 = e^{-\frac{nt}{m}} t \). Each other equation \( i \) depends on \( f_{i-1}(t) \) and \( f_i(t) \). Finally, for \( f_h(t) \), we use the fact that \( \sum_{i=0}^{h} f_i = 1 \) and substitute all the previous solutions. The resulting values are

\[
    f_i(t) = \begin{cases} 
        \frac{1}{n} \left( \frac{n}{m} \right)^i e^{-\frac{nt}{m}} t 
        & i < h \\
        1 - \sum_{k=0}^{h-1} \frac{1}{k!} \left( \frac{n}{m} \right)^k e^{-\frac{nt}{m}} t 
        & i = h 
    \end{cases} 
\]

(4.4)

Note that the solution is the Poisson distribution with \( \lambda = \frac{nt}{m} \). This is no surprise, due to fact that at a given time \( t \), the total number of mapped elements into a specific bucket is distributed according to Bin\((nt, \frac{1}{m})\), thus the corresponding limit distribution is Poisson\((\lambda = \frac{nt}{m})\).

We define the *overflow fraction at time* \( t \) as the fraction of all \( n \) elements that has not been inserted into the buckets by time \( t \) and denote it \( \gamma_{\text{SIMPLE}}(t) \). Thus, \( \gamma_{\text{SIMPLE}}(t = 0) = 1 \), since at the start no elements have been inserted yet. Then, the \( \gamma_{\text{SIMPLE}}(t) \) function is decreasing as more elements are inserted, until it reaches the final overflow fraction \( \gamma_{\text{SIMPLE}} = \gamma_{\text{SIMPLE}}(t = 1) \). Using the solutions above, right before the \( j \)-th elements is hashed, the overflow fraction at time \( t = \frac{j}{n} \) is

\[
    \gamma_{\text{SIMPLE}}(t) = 1 - \frac{m}{n} \sum_{i=0}^{h-1} \frac{1}{i!} \left( \frac{n}{m} \right)^i e^{-\frac{nt}{m}} t \\
    = \frac{m}{n} - \frac{m}{n} \cdot h \cdot \left( 1 - \sum_{k=0}^{h-1} \frac{1}{k!} \left( \frac{n}{m} \right)^k e^{-\frac{nt}{m}} t \right) \\
    = 1 - \frac{m}{n} h + \frac{m}{n} \sum_{i=0}^{h} (h-k) \cdot \left( \frac{n}{m} \right)^{h-k} \cdot \frac{1}{i!} e^{-\frac{nt}{m}} t 
\]

(4.5)
Alternatively, one can also consider the cumulative overflow fraction at time \( t \); namely, considering only the \( n \cdot t \) elements that are handled by this time (and normalizing according to \( n \cdot t \) and not \( n \)). This cumulative overflow fraction is:

\[
\gamma_{\text{simple}}^t(t) = 1 - \frac{m}{n t} \sum_{i=0}^{h-1} \frac{1}{i!} \left( \frac{n}{m} t \right)^i e^{-\frac{n}{m} t} - \frac{m}{n t} \cdot h \cdot \left( 1 - \sum_{k=0}^{h-1} \frac{1}{k!} \left( \frac{n}{m} t \right)^k e^{-\frac{n}{m} t} \right)
\]

At time \( t = 1 \), we get \( \gamma_{\text{simple}} = \gamma_{\text{simple}}(1) = \gamma_{\text{simple}}^t(1) \), which is the overflow fraction of the scheme.

The equations above are only true as long as the average number of memory accesses per element is at most \( a \). Since this average number equals \( t \) at time \( t \), the process is stopped at \( t = a \). Then, the optimality of the scheme is obtained by substituting \( c = \frac{n}{m} \) and \( t = a \) in Equation (4.5) and comparing it to \( \gamma_{\text{LB}}(a) \) of Theorem 1.

\[\square\]

### 4.4.3 Case Study - A Simple \( \langle 1, 1, c, 1 \rangle \) Hashing Scheme

In the simple scheme, each element is hashed exactly once, therefore, both \( d \), the maximum number of memory accesses, and \( a \), the average number of memory accesses, equal to 1. We want to consider the case where the bucket size is also 1, and derive that \( f_0(t) = e^{-\frac{n}{m} t} = e^{-c t} \), and \( f_1(t) = 1 - f_0(t) \).

Next, the overflow fraction is given by \( \gamma_{\text{simple}}(t) = 1 - \frac{m}{n} \cdot \left( 1 - e^{-\frac{n}{m} t} \right) \). Therefore, for \( a = 1 \), it meets the lower bound at \( t = 1 \), using \( c = \frac{n}{m} \), we get \( \gamma_{\text{simple}}(t = 1) = \gamma_{\text{LB}}(a = 1) = 1 - \frac{1}{e} \cdot (1 - e^{-c}) \). And if \( c = 1 \), we get that \( \gamma_{\text{simple}}(t = 1) = e^{-1} = 36.8\% \).

### 4.4.4 Simulation Results

We now compare the analytical results of the simple scheme with simulation results. Since the simple scheme uses a hash function with uniform distribution, we simulated it by successively choosing a bucket for each element uniformly at random. We used a load \( c = 1 \), a bucket size \( h = 3 \), and \( m = 10,000 \) buckets.

Fig. 4.5(a) shows how the bucket occupancies evolve over time. All buckets are empty at the beginning, while at the end, 57.68% of the buckets are full, i.e. hold three elements, 22.40% hold two elements, 14.94% hold a single element, and 4.98% of the buckets are empty. For all functions, the fluid model appears to closely match simulations.

Fig. 4.5(b) shows how the overflow fraction and the cumulative overflow fraction evolve over time. As previously explained, the overflow fraction is a monotonically-decreasing function that starts at 1, while the cumulative overflow fraction is a monotonically-increasing function that starts at 0. Both functions get the same value at the end. Here again, for
Figure 4.5: Model and simulation results for the SIMPLE scheme given load $c = 1$, bucket size $h = 3$, and memory size $m = 10,000$. 

(a) Fraction of buckets that store $i \in \{0, \ldots, 3\}$ elements.

(b) Overflow fraction.
both functions, our fluid model appears to closely match simulations.

4.5 GREEDY - A Multiple-Choice Hashing Scheme

In the GREEDY scheme, we use an ordered set of $d$ hash functions $H = \{H_1, \ldots, H_d\}$, such that all the hash functions are independent and uniformly distributed. Upon inserting an element $x$, the scheme successively reads the buckets $H_1(x), H_2(x), \ldots, H_d(x)$ and places $x$ in the first non-full bucket. If all these buckets are full, $x$ is placed in the overflow list. Last, as in SIMPLE, to keep an average number of memory accesses per element of at most $a$, the process stops when a total of $a \cdot n$ memory accesses is reached, and the remaining elements are placed in the overflow list as well.

Fig. 4.6 illustrates GREEDY with $m = 12$, $h = 1$ and $d = 2$. We can see that element 6 is initially mapped by $H_1$ to a full bucket. It is therefore mapped again by $H_2$, and inserted in an empty bucket. On the contrary, element 7 is directly inserted in an empty bucket, and therefore does not need a second memory access.

4.5.1 Description by Differential Equations

We model the dynamics of the GREEDY scheme as a system of differential equations, in which time is scaled according to element arrivals. As before, let $f_i(t)$ represent the fraction of buckets storing $i$ elements at time $t$, then

$$\frac{df_i(t)}{dt} = \begin{cases} -\frac{n}{m} f_0(t) g(t) & i = 0 \\ \frac{n}{m} f_{i-1}(t) g(t) & i = h \\ \frac{n}{m} (f_i(t) - f_{i-1}(t)) g(t) & \text{otherwise} \end{cases}$$  \hspace{1cm} (4.7)$$

where $g(t) = \sum_{k=0}^{d-1} f_h(t)^k = \frac{1-f_h(t)^d}{1-f_h(t)}$, with $f_0(0) = 1$ and $f_i(0) = 0$ for each $i \neq 0$ as an initial condition. Compared to the differential equations of the SIMPLE scheme from Equation (4.3), there is an additional factor $g(t)$. For instance, in the first equation, $f_0(t)$ is replaced by $f_0(t) g(t) = \sum_{k=0}^{d-1} \left[f_h(t)^k \cdot f_0(t)\right]$, which represents the sum of the probabilities of entering an empty bucket after $k = 0, 1, \ldots, d - 1$ hits at full buckets.
The process stops when reaching a total of \( a \cdot n \) memory accesses, thus we keep count of the total number of memory accesses. Let \( f_{a, \text{GREEDY}}^a(t) \) denote the cumulative number of memory accesses done by time \( t \), normalized by \( n \). It can be modeled as

\[
\frac{df_{a, \text{GREEDY}}^a(t)}{dt} = \sum_{k=1}^{d-1} k \cdot (h_k(t))^{k-1} (1 - f_h(t)) + d \cdot (f_h(t))^{d-1},
\]  

(4.8)

with \( f_{a, \text{GREEDY}}^a(0) = 0 \) as an initial condition. We stop the process when either \( t = 1 \) or \( f_{a, \text{GREEDY}}^a(t) \) reaches \( a \). The differential equation reflects the fact that at a given time \( t \), the cumulative number of memory accesses increases by \( 1 \leq k < d \) memory accesses whenever the first \( k-1 \) memory accesses hit full buckets and the next one hits a non-full bucket. It also increases by \( d \) memory accesses whenever the first \( d-1 \) memory accesses hit full buckets, independently of the bucket state in the \( d \)-th memory access.

### 4.5.2 Optimality of the GREEDY Scheme

We now want to show the optimality of the GREEDY scheme over a range of values of \( a \). In general, the above differential equations are hard to solve analytically, and thus cannot help in showing optimality — even though they can of course be solved numerically and yield a numerical approximation of the expected overflow fraction.

Instead, to show the optimality of the GREEDY scheme, we reduce it to the optimality of the SIMPLE scheme using the cloning method. Since both the SIMPLE and GREEDY schemes use the same uniform distribution, a new attempt to insert an element after hitting a full bucket in the GREEDY scheme is equivalent to creating a new element (or clone) in the SIMPLE scheme and then trying to insert it. In other words, the number of clones successfully inserted by the GREEDY scheme after considering \( n \) elements and using a total of \( a \cdot n \) memory accesses is the same as the number of elements successfully inserted by the SIMPLE scheme after considering \( a \cdot n \) clones and using a single memory access per clone.

We next show that GREEDY is an optimal \( (a, d, c, h) \) hashing-scheme for \( a \leq f_{a, \text{GREEDY}}^a(1) \) and any values of \( d, c \) and \( h \). We call the value \( f_{a, \text{GREEDY}}^a(1) \) the cut-off point of the GREEDY scheme and denote it by \( a^\text{CO, GREEDY} \); beyond this average number of memory accesses per element, the GREEDY scheme is not necessarily optimal anymore. It is important to notice that the differential equations, described in Section 4.5.1, are used only to obtain the optimality range (that is, to calculate the value of \( a^\text{CO, GREEDY} \)).

The following theorem summarizes this.

**Theorem 5** The GREEDY scheme solves the optimal hash table construction problem for \( a \leq a^\text{CO, GREEDY} \) and any values of \( d, c \) and \( h \), where \( a^\text{CO, GREEDY} = f_{a, \text{GREEDY}}^a(1) \).

**Proof.** We compare the GREEDY scheme with the SIMPLE scheme. In the GREEDY scheme, we continually try to insert each element, until either it is placed or all \( d \) functions are used. Note that all hash functions have the same (uniform) distribution over all buckets. Thus, for every \( i \), \( f_i(t) \)—and therefore also \( \gamma_{\text{GREEDY}}(t) \)—are independent of the exact elements that
are hashed. Moreover, applying \( d_1 \leq d \) hash functions on the same element is equivalent to applying a single hash function on \( d_1 \) different elements. This implies that the results of the SIMPLE scheme hold also in this case with a different time scale: The insertion process continues beyond time 1, until time \( a \), in which a total of \( a \cdot n \) memory accesses are performed. Thus, the overflow fraction is obtained by substituting \( t = a \) in Equation (4.6).

Note, however, that the total number of memory accesses after all elements are considered is given by \( f_a^g \text{GREEDY} (1) \). Thus, if \( a \) is larger than \( f_a^g \text{GREEDY} (1) \) it does not impose an effective restriction on the scheme; this, in turn, implies that by increasing \( a \) beyond \( f_a^g \text{GREEDY} (1) \) one cannot improve the scheme performance.

Hence, the GREEDY scheme solves the OPTIMAL HASH TABLE CONSTRUCTION PROBLEM for \( a \in [1, a^co \text{GREEDY}] \), because both expressions are identical and GREEDY does not perform more than \( a^co \text{GREEDY} \) memory accesses per insertion. Therefore, the limit overflow fraction as a function of the average memory accesses is given by:

\[
\gamma_{\text{GREEDY}} (a) = 1 - \frac{1}{c} + \frac{1}{ch} \sum_{i=0}^{h} (h - i) \cdot \frac{(aech)^i}{i!} e^{-aech},
\]

where \( a_e = \min \{a, f_a^g \text{GREEDY} (1)\} \). \( \square \)

4.5.3 Case Study - A GREEDY \( \langle a, 2, c, 1 \rangle \) Hashing-Scheme

Although in general it is difficult to obtain analytically the value of \( a^co \text{GREEDY} \) for general values of \( h \) and \( d \), in the following section we describe how it can be obtained when \( h = 1 \) and \( d = 2 \).

In this specific case, \( f_0 (t) + f_1 (t) = 1 \) and therefore the first differential equation in (4.7) is

\[
\frac{df_0(t)}{dt} = -\frac{n}{m} \left( 2f_0 (t) - (f_0 (t))^2 \right),
\]

where \( f_0(0) = 1 \) is the given initial condition. Since this is a Bernoulli differential equation, it can be solved analytically. So, \( f_0 (t) = \frac{2}{1 + e^{\frac{2}{m} t}} \) and \( f_1 (t) = -\frac{1 + e^{\frac{2}{m} t}}{1 + e^{\frac{2}{m} t}} \). The overflow fraction is given by

\[
\gamma_{\text{GREEDY}} (t) = 1 - \frac{m - 1 + e^{\frac{2}{m} t}}{1 + e^{\frac{2}{m} t}} \quad (4.9)
\]

Finally, in order to compute \( f_a^g \text{LEGREEDY} (t) \), we substitute \( f_0 (t) \) and \( f_1 (t) \) in Equation (4.8) and solve the resulting differential equation:

\[
\frac{df_a^g \text{GREEDY}(t)}{dt} = \frac{2e^{\frac{2}{m} t}}{1 + e^{\frac{2}{m} t}}.
\]

Integrating the right side and applying the initial condition yields:

\[
f_a^g \text{GREEDY} (t) = \frac{m}{n} \cdot \ln \left( 1 + e^{\frac{2}{m} t} \right),
\]

and by Theorem 5, the range of optimality in this specific case is \( a \in \left[ 1, \frac{n}{m} \cdot \ln \left( 1 + e^{\frac{2}{m} t} \right) \right] \).

In particular, if \( n = m \), the cut-off point is \( a^co \text{GREEDY} = \ln \left( \frac{e^{2} + 1}{2} \right) \approx 1.4338 \) and the corresponding overflow fraction is \( \gamma_{\text{GREEDY}}(t = 1) = \frac{2}{e^{2} + 1} \approx 23.8\% \). Likewise, if \( n = 0.1m \), the cut-off point is \( a^co \text{GREEDY} = 10 \ln \left( \frac{e^{0.2} + 1}{2} \right) \approx 1.0499 \) and the corresponding overflow fraction
is \( \gamma_{\text{greedy}}(t = 1) = 1 - 10 \cdot \frac{e^{-0.2}}{e^{0.2} + 1} \approx 0.33\% \).

In case \( a \) is within the range of optimality, by solving the equation \( f_{a,\text{greedy}}'(t) = a \), we can obtain the time at which the bound \( a \) is reached and no more memory accesses are allowed. In our case this time turns to be \( t = \frac{1}{2} \cdot \frac{m}{n} \cdot \ln \left(2e^{\frac{a}{m}} - 1\right) \).

Note that by substituting \( t \) in Equation (4.9), we get:

\[
\gamma_{\text{greedy}}(a) = 1 - \frac{m}{n} - e^{\frac{2}{n} \cdot \ln \left(2e^{\frac{a}{m}} - 1\right)}
\]

which matches exactly the result in Theorem 5.

### 4.6 The Multi-Level Hash Table (MHT) Scheme

In this section, we consider another hashing scheme, the multi-level hash table (MHT), presented in Example 1. We show that it is optimal with respect to the average number of memory accesses \( a \) also beyond \( a_{\text{greedy}}^\text{opt} \), the value below which the greedy scheme is proven to be optimal (Theorem 5). Thus, the MHT scheme improves upon the greedy scheme. We later compare in details the performance of these schemes in Section 4.8.

Recall that in MHT, hash function \( H_i \) maps to subtable \( T_j \) (of size \( \alpha_i \cdot m \)), that is, each of the hash functions maps to a different subtable. Therefore, each of the hash functions has a different distribution. Theorem 3 states that in this case, the overflow fraction lower bound \( \gamma_{\text{LB}} \) is computed using a weighted average distribution that is uniform across all buckets. As we later show, the MHT scheme implicitly complies with this condition when the subtable sizes follow a specific geometric decrease.

#### 4.6.1 Description by Differential Equations

The system of differential equations that characterizes the dynamics of MHT is similar to that of the greedy scheme, although the static partitioning of the memory among subtables introduces extra variables. Specifically, let \( f_{i,j}(t) \) be the fraction of buckets in subtable \( T_j \) that store exactly \( i \) elements. Then:

\[
\frac{df_{i,j}(t)}{dt} = \begin{cases} 
-\frac{n}{\alpha_j m} f_{0,j}(t) g_{j}(t) & i = 0 \\
\frac{n}{\alpha_j m} f_{i-1,j}(t) g_{j}(t) & i = h \\
\frac{n}{\alpha_j m} (f_{i-1,j}(t) - f_{i,j}(t)) g_{j}(t) & \text{otherwise}
\end{cases}
\] (4.10)

where \( g_{j}(t) = \prod_{k=1}^{j-1} f_{h,k}(t) \) represents the probability that all the insertion attempts in subtables \( T_1, \ldots, T_{j-1} \) meet full buckets, and thus MHT attempts to insert the element in subtable \( T_j \). By convention, \( g_1(t) = 1 \); the initial conditions are \( f_{i,j}(0) = 1 \) for \( i = 0 \) and \( f_{i,j}(0) = 0 \) otherwise.
As in the greedy scheme, let $f_{\text{MHT}}^a(t)$ denote the cumulative number of memory accesses done by time $t$, normalized by $n$. Then the following differential equation reflects the dynamics of $f_{\text{MHT}}^a(t)$:

$$
\frac{df_{\text{MHT}}^a(t)}{dt} = \sum_{k=1}^{d-1} k \cdot g_k(t) (1 - f_{h,k}(t)) + d \cdot g_d(t),
$$

with $f_{\text{MHT}}^a(0) = 0$.

### 4.6.2 Reduction to the Simple Scheme

As in the greedy scheme, we prove the optimality of the MHT scheme by reducing it to the simple scheme, and do not rely on the differential equations, which are hard to solve analytically.

Our approach relies on the fact that each subtable follows a local simple scheme. More specifically, all elements attempting to access some subtable $T_j$ only access a single uniformly-distributed bucket in $T_j$, and if this bucket is full, never return to $T_j$. Thus, within each subtable $T_j$, MHT behaves like the simple scheme, with a number of initial elements that depends on previous subtables.

Formally, let $n_j(t)$ denote the number of elements that are considered in subtable $T_j$ up to time $t$, and $\gamma_j^t(t)$ denote the fraction of these elements that are not placed in subtable $T_j$. We will express these using $f_i^\text{simple}$ and $\gamma_i^\text{simple}$, the corresponding functions in the simple scheme. Note that, as shown in Equations (4.4) and (4.6), $f_i^\text{simple}(t)$ and $\gamma_i^\text{simple}(t)$ depend only on the time $t$, the number of elements $n$, the number of buckets $m$, and the bucket size $h$; thus, we refer to them as $f_i^\text{simple}(1, \alpha_j m, n_j(t), h)$ and $\gamma_i^\text{simple}(1, \alpha_j m, n_j(t), h)$. We obtain the following theorem, which is valid for any partition of the subtables.

**Theorem 6** Consider an $\langle a, d, c, h \rangle$ MHT hashing scheme in which for each $1 \leq j \leq d$, subtable $T_j$ has $\alpha_j \cdot m$ buckets, with $\sum \alpha_j = 1$. Then, as long as $f_{\text{MHT}}^a(t) \leq a$,

(i) $n_j(t) = n \cdot t \cdot \prod_{k=1}^{j-1} \gamma_k(t)$,

(ii) $\gamma_j^t(t) = \gamma_j^\text{simple}(1, \alpha_j m, n_j(t), h)$, and

(iii) $f_{i,j}(t) = f_i^\text{simple}(1, \alpha_j m, n_j(t), h)$.

In addition, if the average number of memory accesses does not reach $a$ by the end of the process, the overflow fraction of MHT is given by $\gamma_{\text{MHT}} = \prod_{j=1}^{d} \gamma_j^t(1)$.

**Proof.** By the definition of the MHT scheme, it follows immediately that $n_j(t) = \gamma_{j-1}^t(t) n_{j-1}(t)$; since $n_1(t) = n \cdot t$ (all elements go through the first subtable), we get that $n_j(t) = n \cdot t \cdot \prod_{k=1}^{j-1} \gamma_k^t(t)$.

The claimed result is immediately derived by setting the right parameters for each simple scheme within each subtable $T_j$; namely, its total number of buckets is $\alpha_j \cdot m$ and the number of elements by time $t$ is $n_j(t)$.
4.6.3 Optimality of the MHT Scheme

We now prove that MHT is optimal on a given range of $a$, and in particular we show that the overflow fraction $\gamma_{MHT}$ of the MHT scheme reaches the overflow fraction lower bound $\gamma_{LB}$ for such $a$. Further, we demonstrate that MHT is optimal when its subtables sizes follow a specific geometric decrease.

**Theorem 7** Let $p(a) = \gamma^t_{\text{SIMPLE}}(1, m, a \cdot n, h)$ denote the overflow fraction of the SIMPLE scheme with $a \cdot n$ elements. Then, the $(a, d, c, h)$ MHT scheme solves the OPTIMAL HASH TABLE CONSTRUCTION PROBLEM whenever it satisfies the two following conditions:

(i) The subtable sizes $\alpha_j \cdot m$ follow a geometric decrease of factor $p(a)$: $\alpha_j = \left(\frac{1-p(a)}{1-p(a)^d}\right) p(a)^{j-1}$;

(ii) $a \leq a^\text{co}_{MHT}$, where $a^\text{co}_{MHT}$ is given by the solution of the following fixed-point equation:

$$a^\text{co}_{MHT} = \frac{1-p(a)^d}{1-p(a)}.$$ 

**Proof.** Consider a specific time $t_0$, in which we exhausted all $a \cdot n$ memory accesses. Up until this time, we used exactly $n_j(t_0)$ times the hash function $H_j(x)$ in subtable $T_j$. Since we aim at an optimal overflow fraction, the necessary condition on the distributions of hash functions, given in Theorem 3, immediately implies that $\alpha_j = \frac{n_j(t_0)}{n \cdot a}$. By substituting the expression for $\alpha_j$ from Theorem 6, we get:

$$\gamma^t_j(t_0) = \gamma^t_{\text{SIMPLE}} \left(1, \frac{n_j(t_0)}{n \cdot a}, m, n_j(t_0), h\right)$$

$$= 1 - \frac{n_j(t_0)}{n \cdot a} \cdot \frac{m h}{n j(t_0)} + \frac{n_j(t_0)}{n \cdot a} \cdot \frac{m_{j}(t_0)}{n \cdot a}$$

$$= 1 - \frac{m h}{n \cdot a} + \frac{m}{n \cdot a} \sum_{i=0}^{h} (h-i) \cdot \frac{1}{i!} \left(\frac{n_j(t_0)}{n_j(t_0) \cdot m}\right)^i e^{-\frac{n_j(t_0)}{n_j(t_0) \cdot m}}$$

$$= 1 - \frac{m h}{n \cdot a} + \frac{m}{n \cdot a} \sum_{i=0}^{h} (h-i) \cdot (\frac{n \cdot a}{m})^i e^{-\frac{n \cdot a}{m}}$$

$$= \gamma^t_{\text{SIMPLE}}(1, m, a \cdot n, h) = p(a).$$

It is important to notice that, quite surprisingly, $\gamma^t_j(t_0)$ does not depend on $j$.

We now obtain the time $t_0$ by observing that $n_j(t_0) = n \cdot t_0 \cdot p(a)^{j-1}$, thus $\alpha_j = \frac{t_0 \cdot p(a)^{j-1}}{a}$. Since $\sum_{k=1}^{d} \alpha_k = 1$, we get $\sum_{k=1}^{d} \frac{t_0 \cdot p(a)^{j-1}}{a} = 1$, and therefore $t_0$ is given by the sum of a geometric series:

$$t_0 = a \left(\frac{1-p(a)}{1-p(a)^d}\right).$$

(4.12)

This, in turn, immediately gives us the claimed memory partition $\alpha_j$.

We now turn to show that the overflow fraction is indeed optimal. Notice that overflowed elements arise in two situations:
1. The elements are rejected by subtable $T_d$. By the time $t_0$, the fraction of overflowed elements out of these which were considered by $T_d$ is $\gamma_d(a) = p(a)$. Furthermore, since the total number of elements considered is $n_d(a)$, we get that the total number of elements which were moved by $T_d$ to the overflow list is $p(a)n_d(t_0) = n \cdot t_0 \cdot p(a)^d - p(a)^{d-1} p(a) = n \cdot t_0 \cdot p(a)^d$.

2. The elements that arrive after time $t_0$ and are rejected without any consideration, because the number of total memory accesses is exhausted. By the definition of $t_0$, the number of such elements is $n - nt_0$.

Hence, the overflow fraction of the MHT scheme with the above memory partitioning is:

$$\gamma_{\text{MHT}}(a) = \frac{n - n \cdot t_0 + n \cdot t_0 \cdot p(a)^d}{n}$$

$$= 1 - t_0 \left(1 - p(a)^d\right) = 1 - a(1 - p(a))$$

$$= 1 - \frac{m h}{n} \cdot \frac{m}{n} \sum_{i=0}^{h} (h - i) \cdot \frac{1}{\alpha^i} \left(\frac{na}{m}\right)^i e^{-\frac{na}{m}}$$

By comparing $\gamma_{\text{MHT}}(a)$ with Theorem 1, we immediately conclude that the MHT scheme with the above partitioning solves the OPTIMAL HASH TABLE CONSTRUCTION PROBLEM for the given $a$.

Finally, we observe in Equation (4.12) that as $a$ increases, the time $t_0$ at which all memory accesses are exhausted also increases. This implies that preserving the tightness to the lower bound can continue only until $a_{\text{MHT}}^\alpha$, when $t_0 = 1$.

In the above theorem, for any $a \leq a_{\text{MHT}}^\alpha$, we found a specific partition of MHT that achieves optimality. Note that the overflow fraction can still be improved beyond $a_{\text{MHT}}^\alpha$, albeit without preserving tightness to the overflow fraction lower bound $\gamma_{\text{LB}}$.

### 4.6.4 Case Study - A MHT $\langle a, 2, 1, 1 \rangle$ Hashing-Scheme

In the following section, we consider a specific MHT scheme with $h = 1$, $d = 2$, and $m = n$ (i.e. with load $c = 1$). For this specific case, we can provide closed-form expressions for the cut-off point and the overflow ratio, by directly solving the set of differential equations in Equation (4.10). In case $h = 1$ and $d = 2$, this set is reduced to:

$$\frac{df_{0,1}(t)}{dt} = -\frac{1}{\alpha_1} f_{0,1}(t) \quad \frac{df_{0,1}(t)}{dt} = -\frac{1}{\alpha_2} f_{0,2}(t) \cdot f_{1,1}$$

$$\frac{df_{0,2}(t)}{dt} = -\frac{1}{\alpha_2} f_{0,2}(t) \cdot f_{1,1} \quad \frac{df_{1,1}(t)}{dt} = \frac{1}{\alpha_2} f_{0,2}(t) \cdot f_{1,1}$$

(4.13)

where $f_{0,1}(0) = f_{0,2}(0) = 1$ and $f_{1,1}(0) = f_{1,2}(0) = 0$.

As in the simple scheme, we immediately get that $f_{0,1}(t) = e^{-\frac{1}{\alpha_1} t}$ and $f_{1,1}(t) = 1 - e^{-\frac{1}{\alpha_1} t}$. By substituting $f_{1,1}$ in Equation (4.13), we get that $f_{0,2}(t) = \exp\left(-\frac{1}{\alpha_2} t - \frac{a_1}{\alpha_2} e^{-\frac{1}{\alpha_1} t} + C_1\right)$, where $C_1 = \frac{a_1}{\alpha_2}$ is the constant determined by the initial
condition. It also immediately follows that \( f_{1,2}(t) = 1 - f_{0,2}(t) \). Thus, the overflow fraction \( \gamma_{\text{MHT}}(t) \) is given by:

\[
\gamma_{\text{MHT}}(t) = \frac{n - (\alpha_1 m f_{1,1}(t) + \alpha_2 m f_{1,2}(t))}{n} = \alpha_1 e^{-\frac{1}{\alpha_1} t} + \alpha_2 e^{\left(-\frac{1}{\alpha_2} t - \frac{\alpha_1}{\alpha_2} e^{-\frac{1}{\alpha_1} t} + \frac{\alpha_1}{\alpha_2}\right)},
\]

In order to compute \( f^a_{\text{MHT}}(t) \), we substitute \( f_{0,1}(t), f_{1,1}(t) \) and \( f_{0,2}(t) \) in Equation (4.11) and solve the resulting differential equation:

\[
\frac{df^a_{\text{MHT}}(t)}{dt} = 2 \cdot \left(1 - e^{\frac{-1}{\alpha_1} t}\right) + 1 \cdot \left(e^{\frac{-1}{\alpha_1} t}\right) = 2 e^{\frac{-1}{\alpha_1} t},
\]

whose solution is \( f^a_{\text{MHT}}(t) = 2t + \alpha_1 e^{\frac{-1}{\alpha_1} t} - \alpha_1 \).

For the case we consider, \( p(a) = 1 - \frac{1}{a} + \frac{1}{a} e^{-a} \). Thus, by Theorem 7, we get the fixed point equation \( a^c_{\text{MHT}} = 2 - \frac{1}{a_{\text{MHT}}} + \frac{1}{a_{\text{MHT}}} e^{-a_{\text{MHT}}}, \) with the positive solution \( a^c_{\text{MHT}} = 1 + 2 \cdot W\left(\frac{1}{2} e^{-\frac{1}{2}}\right) \approx 1.4777, \) where the Lambert W function is the inverse function of the function \( \omega(x) = xe^x \) [97].

At the cut-off point, assuming an optimal partition, the overflow fraction is \( \gamma_{\text{MHT}}(t = 1) = e^{-a^c_{\text{MHT}}} \approx 22.8\%. \) Likewise, if \( n = 0.1m \), the cut-off point is \( a^c_{\text{MHT}} = 1.0507 \) and the corresponding overflow fraction is \( \gamma_{\text{MHT}}(t = 1) = 0.26\% \).

### 4.7 Dealing With Non-Unique Elements

Up until now, we have considered the case where all \( n \) elements are unique (and have a single instance). In this section, we consider the case where elements are unnecessarily unique. An example application is where the value associated with each element stored at the hash table need to be updated upon arrival of every instance of each such element.

In the case where some of the elements are alike, the models proposed in previous sections do not apply; when a new element arrives, the hash functions used are independent of previous elements hash values, which is clearly not the case when a second instance of a previously arrived element arrives.

Nevertheless, in both the Greedy and the MHT schemes, whenever an instance of an existing element arrives, it needs as many memory accesses as the first one. Therefore, since the expected number of memory accesses for each element increases over time in the case of unique elements, the arrival order affects the overall average number of memory accesses. For example, if the first instance of a highly frequent element arrives first (among all elements), then all future instances of this element will require only one memory access, while if the first instance arrives later, it may need more memory accesses, resulting in a larger average number of memory accesses.

We first provide an approximate model in Section 4.7.1 by assuming that the exact order
of arrival is given. In Section 4.7.2, we relax this assumption and assume that we are given a histogram of the number of unique elements per number of instances. This histogram corresponds to the flow size distribution. In Section 4.8.3, we compare our model with trace-driven simulations.

### 4.7.1 Model Given the Arrival Order of Elements

Consider a sequence $S$ of $N$ non-unique elements instances that arrive one after the other, where the sequence consists of $n$ unique elements $\mathcal{U} = \{e_1, \ldots, e_n\}$, sorted in the order of the arrival of the elements’ first instance (that is, for every $i < j$, the first instance of $e_i$ arrives before the first instance of $e_j$). For each element $e_i \in \mathcal{U}$, let $s_i$ be the number of times $e_i$ appears in $S$. Note that $\sum_{i=1}^{n} s_i = N$.

Upon arrival of an element instance, the hash scheme first checks whether the element already exists in the hash table. In particular, both the greedy and mht schemes access buckets using their ordered set of the hash functions either up until the element is found or until a non-full bucket is accessed, implying that the element does not exist in the hash table. If all $d$ buckets are full and the element is not found, the scheme must check the overflow list before inserting the element.

Thus, in both the greedy and mht schemes, a non-first insertion of some element results in the same sequence of memory operations as in the first insertion of the element. Since a non-first insertion of a specific element does not affect the overflow fraction, then the overflow fraction of both the greedy and mht schemes can be computed as in the previous sections, using the $n$ unique elements instead of the $N$ non-unique element instances.

However, the average number of memory accesses is potentially changed. Let $t_j = \frac{4}{n}$ be the normalized time at which element $e_j$ appears in the sequence $\mathcal{U}$. In both the greedy and mht schemes, an arrival of an already-existing element takes exactly the same number of memory accesses as it did in its first insertion. Therefore, the average number of memory accesses can be modeled as a weighted sum of the number of memory accesses that each element insertion causes, where the weight of each element $e_j$ is given by its number of current and future instances $s_j$. Thus, $a \approx \frac{n}{N} \cdot \sum_{j=0}^{n} s_j \cdot f^{\mathcal{X}}_X (t_j) \cdot \frac{4}{n}$, where $f^{\mathcal{X}}_X (t)$ denotes the cumulative number of memory accesses done by time $t$, normalized by $n$, and $X$ is the hashing scheme (greedy or mht). Therefore, $f^{\mathcal{X}}_X (t_j)$ gives the number of memory accesses performed at time $t_j$.

Note that if all elements are unique, i.e. $s_j \equiv 1$ for all $j$, then $N = n$ and the above expression reduces to

$$a \approx \sum_{j=0}^{n} f^{\mathcal{X}}_X (t_j) \cdot \frac{1}{n} \approx \int_{0}^{1} f^{\mathcal{X}}_X (t) \cdot dt = f^{\mathcal{X}}_X (1),$$

where the last equality follows from the fact that $f^{\mathcal{X}}_X (0) = 0$. Thus, we conclude that the same model of the previous sections applies also for this case.
4.7.2 Model Given the Instances Histogram

In the last section, we found that the average number of memory accesses depends only on the arrival order of the first element instances and on the number of instances $s_j$ of the $j$-th arriving unique element. In this section, we model the expected number of instances $s_j$ of the $j$-th arriving unique element given the histogram of the number of unique elements per number of instances. For instance, we might be given a general model for the flow size distribution, without a specific arrival order of precise element instances.

Intuitively, the first instance arrival of a more frequent element is expected to be relatively early, and thus the element is expected to be located in the hash table in one of its first choices. Thus, the average number of memory accesses is expected to decrease with respect to the case where each unique element arrives exactly once.

Formally, given the number of unique elements $n$ and the number of instances of each element, we construct a vector (histogram) $B$ of size $b$ in which the $i$-th element corresponds to the number of unique elements that have exactly $i$ instances. Our model assumes that at any given time the probability that some unique element arrives next is proportional to its remaining number of instances (according to the histogram).

We extract the number of instances $s_j$ of the $j$-th arriving unique element $e_j$ iteratively, where the $j$-th iteration corresponds to the $j$-th unique element arrival. At each step, we find the probability that the next arriving unique element has exactly $x$ instances. Then, we compute the expected number of instances out of that distribution. Finally, we update the histogram towards the next step by subtracting from the histogram the computed distribution. Let $\Phi(\cdot)$ be an operator that multiplies each entry in its input vector by its index. For example, by applying $\Phi(\cdot)$ on the vector $[3, 2, 4]$, we get $\Phi([3, 2, 4]) = [3 \cdot 1, 2 \cdot 2, 4 \cdot 3] = [3, 4, 12]$. Also, set $B_1 = B$, we get:

$$s_j = \frac{\Phi(B_j)}{\|\Phi(B_j)\|_1}, \quad B_{j+1} = B_j - \frac{\Phi(B_j)}{\|\Phi(B_j)\|_1},$$

where $\|\cdot\|_1$ is the L1-Norm. Note that $\frac{\Phi(B_j)}{\|\Phi(B_j)\|_1}$ gives a probability vector in which the $x$ entry corresponds to the probability that the next arriving unique element has exactly $x$ instances.

Note that this model summarizes at each step the expected histogram. Therefore, it is only an approximation which is more accurate for a large number of (unique) elements. We illustrate this model in the following example.

**Example 6** Let $B = [3, 1]$, which means that there are three elements that appears once and one element that appears twice, yielding a total of five instances. We first set $B_1 = B$ and compute the distribution of the number of instances of the first arriving element $e_1$. It is given by $\frac{\Phi(B_1)}{\|\Phi(B_1)\|_1} = [\frac{3}{5}, \frac{2}{5}]$. This means that the element that has two instances arrives first with probability of $\frac{2}{5}$, and one of the other three elements (with one instance) arrives first with probability of $\frac{3}{5}$, that is, a probability of $\frac{1}{5}$ each. Therefore, the expected number of instances of the first arriving unique element $e_1$ is given by $\frac{2}{5} \cdot 2 + \frac{3}{5} \cdot 1$, that is, as given
by the model, $s_1 = \| \Phi \left( \left[ \frac{3}{5}, \frac{2}{5} \right] \right) \|_1 = \frac{7}{5}$.

We now update the histogram for the next iteration and get that $B_2 = B_1 - \frac{\Phi(B_1)}{\| \Phi(B_1) \|_1} = [3, 1] - \left[ \frac{3}{5}, \frac{2}{5} \right] = \left[ \frac{12}{5}, \frac{3}{5} \right]$. Continuing this process we get that $s_2 = \frac{4}{3}$, $s_3 = \frac{24}{17}$ and $s_4 = \frac{279}{263}$.

4.8 Comparative Evaluation

4.8.1 Lower bound vs. GREEDY and MHT

Fig. 4.7 illustrates the influence of the memory partition on the overflow fraction and the optimality of MHT. It was obtained with $d = 4$, $h = 4$ and $c = 1$. All values were derived from the analytical formulas above, except for the d-left hashing scheme, for which we ran simulations with $m = 4,000$, $n = 16,000$ and $d$ equally-sized subtables.

First, the solid line plots the overflow fraction lower-bound $\gamma_{\text{LB}}(a)$ from Theorem 1. Thus, no scheme can achieve an asymptotic overflow fraction below this line.

As elements are successively inserted and the total number of memory accesses $a \cdot n$ increases, the overflow fractions $\gamma_{\text{SIMPLE}}(a)$ and $\gamma_{\text{GREEDY}}(a)$ of the SIMPLE and the GREEDY schemes follow this lower-bound line, respectively until $a_{\text{SIMPLE}}^{\triangle} = 1$ with $\gamma_{\text{SIMPLE}} = 19.5\%$ (Theorem 4), and $a_{\text{GREEDY}}^{\triangle} = 1.488$ with $\gamma_{\text{GREEDY}} = 6.00\%$ (Theorem 5).

On the contrary, in the case of MHT, for a given partition, $\gamma_{\text{MHT}}(a)$ does not go down along the lower-bound line. As shown in the proof of Theorem 7, for any given $a$, an MHT scheme using the optimal geometrically-descending partition for $a$ will be strictly above the lower-bound line, then reach it at $a$, then rebound and be above it again. This is indeed illustrated using the optimal partitions for $a = 1$ and $a = a_{\text{MHT}}^{\triangle} = 1.697$. The corresponding optimal overflow fractions are $\gamma_{\text{MHT}}(a = 1) = \gamma_{\text{SIMPLE}} = 19.5\%$ and $\gamma_{\text{MHT}}(a = a_{\text{MHT}}^{\triangle}) = 3.45\%$. 

Figure 4.7: Overflow fraction as a function of $a$ with $d = 4$, $h = 4$, $c = 1$. OP$(a)$ denotes the optimal partition of MHT for $a$ as obtained by Theorem 7.
Last, we compare the performance of MHT with that of the \(d\)-left algorithm, in which \(a = d = 4\) [25, 26]. It can be seen that \(d\)-left achieves an overflow fraction of \(\gamma_{d\text{-left}} = 3.17\%\), which is not far from the overflow fraction of MHT with \(a = a_{\text{MHT}}^{co}\), while on average MHT saves more than half of the memory accesses.

### 4.8.2 Effects of the System Parameters

Fig. 4.8(b) compares the respective cut-off points \(a_{\text{GREEDY}}^{co}\) and \(a_{\text{MHT}}^{co}\) of GREEDY and MHT under different loads, with \(h = 4\) and \(d = 4\). Clearly, the cut-off point of MHT is higher, implying that its range of optimality is larger. Additionally, Fig. 4.8(a) shows the corresponding overflow fractions \(\gamma_{\text{GREEDY}} (a = a_{\text{GREEDY}}^{co})\) and \(\gamma_{\text{MHT}} (a = a_{\text{MHT}}^{co})\), illustrating that MHT can achieve a lower overflow fraction.

Figure 4.9 compares the cut-off points \(a = a_{\text{GREEDY}}^{co}\) and \(a = a_{\text{MHT}}^{co}\) of GREEDY and MHT under different maximum numbers of memory accesses, where \(h = 4\) and \(c = 0.9\). Specifically, Figure 4.9(b) shows that the cut-off point of MHT is larger than the cut-off point of GREEDY, where both cut-off points tend to 1.429. This is the point where the lower bound provided in Section 4.3 reaches an overflow fraction of 0. Consequently, as the cut-off points tend to this value at a different rate, so do the corresponding overflow fractions shown in Figure 4.9(a). Although there is only a minor difference in the cut-off points, the corresponding overflow fraction of MHT is much lower than the corresponding overflow fraction of GREEDY, illustrating the strength of the MHT scheme using the provided optimal partition of the memory, as found in Section 4.6.
4.8.3 Trace-Driven Simulation With Non-Unique Elements

We have also conducted experiments of both the GREEDY and MHT schemes using real-life traces recorded on a single direction of an OC192 backbone link [98]. Our goal is to validate our model using real hash functions; we used a 64-bit mix function [99] that was subdivided into four distinct 16-bit hash functions.

We took \( m = 2^{14} \), \( h = 4 \), \( d = 4 \), and set \( n \) corresponding to various values of the load \( c \). Each simulation was run up until exactly \( n \) unique elements arrive, where each element corresponds to the standard 5-tuple (srcIP, dstIP, srcPort, dstPort, protocol) of each packet in the trace. Finally, for the MHT scheme, we picked the memory partition that is optimal for the average number of memory accesses \( a_{\text{co}}^{\text{MHT}} \), as in Theorem 7.

Figure 4.10 shows the average number of memory accesses found by simulations as well as by our proposed model from Section 4.7.2 which assumes that only the histogram of number of unique elements per number of instance is given but the exact order of arrival is not. For comparison, we also add the expected number of memory accesses when each unique element appears only once. The simulation shows that the non-unique model is fairly accurate. In the worst case, when the load is 1, the average number of memory accesses using the GREEDY (MHT) scheme was found by the simulation to be 1.1649 (1.2680) when our model predicts 1.1266 (1.2039), compared to a modeled value of 1.4887 (1.6970) when each unique element arrives only once.

As expected from the model, the arrival of non-unique elements decreases the average number of memory accesses. However, there is a minor disparity between the simulation and the model with non-unique elements. It is because packets of the same flow tend to arrive close to each other in the trace, and are not uniformly distributed over the whole
Figure 4.10: Average number of memory accesses with non-unique elements

Figure 4.11: Overflow fraction with non-unique elements
trace. Consequentially, it decreases the probability of the first packet in a large flow to arrive at the start of the trace.

We also note that the GREEDY scheme uses less memory accesses than the MHT scheme. Intuitively, this is because elements in the GREEDY scheme have better chances of being inserted in the first bucket(s) than in the MHT scheme; more formally, $f_{\text{GREEDY}}(t) < f_{\text{MHT}}(t)$ for small values of $t$.

However, this decrease in the number of memory accesses comes with a price. Fig. 4.11 shows the overflow fraction found via the trace-driven simulations and the model. The overflow fraction of the MHT scheme is lower than the one of the GREEDY scheme. For example, for the case where $c = 1$, the overflow fraction of the MHT scheme is 3.56%, compared to 6.07% of the GREEDY scheme. Moreover, as found in Section 4.7, the overflow fraction depends only on the number of unique elements $n$, which is also confirmed in Fig. 4.11.
Chapter 5

Dynamic Table Model

Networking devices often use dynamic hash tables, in which elements keep being both inserted and deleted as packets arrive and depart. Therefore, they differ from static ones, which are built only once. However, for simplicity, device designers typically model the performance of the dynamic hash tables using models of the static hash tables. This chapter shows that these static models can lead to a significant under-estimation of the drop rate in the dynamic case.

While former studies have found the same asymptotic behavior in dynamic and in static hash tables (see Section 2.1.4), in this chapter, we show that when memory is bounded, i.e. memory buckets are finite, dynamic hash tables that allow insertions and deletions behave significantly worse than their static counterparts that only allow insertions. This behavior differs from previous results in which, when memory is unbounded, the two models behave similarly.

We show the decrease in performance in dynamic hash tables using several hash-table schemes. We also provide tight upper and lower bounds on the achievable overflow fractions in these schemes. Finally, we propose an architecture with content-addressable memory (CAM), which mitigates this decrease in performance.

We focus on the realistic scenario in which buckets are finite, as used in networking devices, contrarily to the infinite-bucket case assumed above. We show that the dynamic hash table can exhibit a significantly worse drop rate than its static counterpart. That is, the rate of elements that cannot be inserted in the hash table is significantly higher.

5.1 Performance Degradation

5.1.1 Simulation results

Fig. 5.1 plots the system average overflow fraction as a function of the load, i.e. the fraction of elements not placed in the buckets as a function of the average number of elements (either
dropped or not dropped) per bucket. Specifically, it shows the average overflow fraction for both a static system, where there are only insertions, and a dynamic system, where we alternate between deletions and insertions, while a fixed load is maintained [24, 28]. To measure the overflow fraction, it relies on an overflow list, called stash, to which new elements are moved when they cannot be inserted in the hash table. Fig. 5.1(a) and 5.1(b) show the overflow fraction of the d-random algorithm with a stash [24] (where \(d = 2\)), and the cuckoo hashing with a stash [42, 43]. The overflow fractions are obtained in simulations using 2048 buckets, \(10^6\) rounds with one random element deletion and one element insertion in each round (for the dynamic case), and a standard pseudorandom number generator to obtain hash values. The overflow fraction in the static case is only measured at the end of each simulation.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Average overflow fraction with 2 hash functions and bucket size 1, using both the static and the dynamic model.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5a.png}
\caption{An example demonstrating the degradation of performance in dynamic hash tables.}
\end{figure}

\footnote{Since in practice, sizing the buckets is according to the width of a single SRAM or DRAM memory word [22], we focus in this chapter on buckets of small sizes (e.g. 1, 2 or 4). However, the results in this chapter are general and consider any arbitrary bucket size.}

\footnote{Simulations with ten times more buckets or rounds yielded nearly identical results.}
of all element insertions, while in the dynamic case it is measured after each cycle and then averaged over cycles.

Both figures clearly show a non-negligible degradation in the overflow fraction of the dynamic system. For instance, the cuckoo hashing scheme with load of 0.6 yields an overflow fraction of 0.62% and 3.02% in the static and dynamic models, respectively. Moreover, while for cuckoo hashing scheme with load of 0.5 the overflow fraction in the static model quickly goes to 0 [35], it does so more slowly in the dynamic case. For instance, for \( m = 1024 \) we get an overflow fraction in the static and dynamic models of 0.05% and 0.44%, where for \( m = 16384 \) we get 0.0012% and 0.0606%, respectively.

In the infinite-bucket case, no overflow list is needed. Thus, to compare the finite-bucket case with the case where buckets are infinite, we define the average overflow fraction for the infinite-bucket case as the probability that an element is not the first one in its bucket. Furthermore, since cuckoo hashing scheme is not defined for infinite buckets, we compare the overflow fraction only under the \( d\)-random scheme. Our simulations show a similar overflow fraction in the following three cases: static scheme with buckets of size one, static scheme with infinite buckets, and dynamic scheme with infinite buckets (the first two curves completely coincide in Fig. 5.1(a), so they appear in the legend as “static”, while the last curve is slightly different due to the nature of the \( d\)-random scheme). On the other hand, there is a significant increase in the overflow fraction when the scheme is dynamic and buckets are finite. In the rest of the chapter, we will evaluate this performance degradation and propose methods to mitigate this problem.

### 5.1.2 Intuition

The intuition behind this difference in behavior is that if the bucket size is bounded, once an element is placed in the overflow list it stays there regardless of whether the corresponding bucket become available later upon deletion. Therefore, the order of the insertion and deletion operations directly affects the performance. This is typically not the case in the unbounded bucket case, and the difference can cause a drastic degradation in the scheme performance.

Fig. 5.2 illustrates this degradation in performance, using the same scenario both for the finite and the infinite bucket sizes. For the case of finite buckets, we assume bucket sizes of 1, an overflow list, and an insertion algorithm that uses only one hash function. We consider the following scenario: Let \( t \) be the time when a new element \( x_1 \) is hashed to a full bucket \( j \) that already stores element \( x_0 \) (step (i) in both Fig. 5.2(a) and Fig. 5.2(b)). If a finite bucket is used, then \( x_1 \) is moved to the overflow list (step (ii) in Fig. 5.2(a)), while in the infinite-bucket case, \( x_1 \) is simply stored in bucket \( j \) (step (ii) in Fig. 5.2(b)). Let \( t' > t \) be the time when element \( x_0 \) is deleted. Assuming that element \( x_1 \) is not deleted before \( t' \), it stays in the overflow list in the finite-bucket case, while in the infinite-bucket case it is stored in bucket \( j \) (step (iii)).

Therefore, in the dynamic case with finite bucket sizes, element \( x_1 \) is in the overflow list for a longer time than in the infinite-bucket case.
list, even though its corresponding bucket \( j \) is empty. This could never happen in the static case (elements are stored in the overflow list only after their corresponding buckets are full, and full buckets cannot become empty). It could also never happen in the dynamic case with infinite buckets (there is no overflow list).

### 5.2 Summary of Our Results

We start by considering a simplistic dynamic scheme with a single hash function. We model this hashing scheme analytically using two different models: a discrete-time Markov chain, and a fluid model with a continuous-time Markov process. We find that this simplistic dynamic scheme performs notably worse than its corresponding static scheme.

Then, we obtain a lower bound on the expected overflow fraction in the dynamic model of any hash-table scheme that uses uniform hash functions. We prove that when the average number of memory accesses per insertion \( a \) increases, the expected overflow fraction can decrease as slowly as \( \Omega(1/a) \) (compared to \( \Omega(e^{-a}) \) in the static case [8]). This indicates that the poor performance of dynamic schemes is fundamental, and is hard to solve by simply using additional memory accesses (or hash functions).

Next, we introduce an online multiple-choice scheme (that is, a scheme that uses multiple hash functions). We demonstrate that this scheme reaches the lower bound and therefore is optimal up to a certain rate of memory access, which depends on the system parameters.

However, due to the slow decrease of the lower bound, optimality may be insufficient for certain applications. Therefore, we suggest changing the assumptions and moving back elements from the overflow list when a bucket becomes available upon deletion. We propose the M-B (Moving-Back) scheme that uses a CAM (Content-Addressable Memory) device that stores the elements along with their hash values. A parallel lookup operation is used once an element is deleted and its bucket becomes non-full. This operation, supported by the CAM, finds an element in the overflow list that can be moved back to the bucket. This scheme is shown to beat the initial lower bound without a CAM.

Finally, we evaluate all proposed schemes using simulations as well as experiments with real hash functions applied on real-life traces.

**Chapter Organization** We start with preliminary definitions in Section 5.3. Section 5.4 presents and analyzes the single-choice SINGLE scheme, while Sections 5.5 and 5.6 provide a lower bound on the expected overflow fraction. Then, in Section 5.7 we present and analyze the multiple-choice MULTIPLE scheme, and in Section 5.8 we present the CAM-based M-B scheme which, upon deletion, moves back elements from the overflow list. Finally, we evaluate all the analytical results in Section 5.9.
5.3 Problem Statement

5.3.1 Terminology and Notations

Like traditional hash tables, our schemes in this chapter should support three basic operations: element insertions, element deletions, and lookups. We call the (infinitely long) sequence of these operations the input sequence of the scheme. Consequently, we sometimes refer to an insertion operation as an arrival of an element, and to a deletion operation as a departure of an element. In this chapter, we focus mostly on a specific input sequence, alternating between departures of a random element (picked uniformly at random) and arrivals of a new element [24, 28].

We focus on the average case behavior of the system, as formally defined below.

Definition 5 The expected overflow fraction is the expected fraction of elements (over time) that are not placed in the buckets, that is, are either placed in the overflow list or simply dropped in case the overflow list is absent.

Our goal is to minimize the expected overflow fraction of the scheme, subject to the (total and average) number of memory accesses. We count as one memory access reading and updating all the elements of a single bucket. This corresponds to the common practice of sizing the bucket size by the width of the memory word. We do not count accesses to the overflow list. We further assume that up to \(d\) buckets can be read in parallel before deciding which one to update, requiring a total of \(d\) memory accesses.

Formally, the hashing scheme and the optimization problem are captured by the following two definitions, where the load \(c\) is the ratio of the total number of elements \(n\) by the total memory size \(mh\): \(c = \frac{n}{mh}\).

Definition 6 When the load is \(c\) and the bucket size is \(h\), an \((a, d, c, h)\) hashing scheme is a scheme with an expected (respectively, maximum) number of memory accesses per element of at most \(a\) (respectively, \(d\)).

Definition 7 The optimal dynamic hash table problem is to find an \((a, d, c, h)\) hashing scheme that minimizes the expected overflow fraction \(\gamma\) as the number of elements \(n\) goes to infinity. Whenever defined, let \(\gamma_{opt}\) denote this optimal expected limit overflow fraction.

5.3.2 Input Models

Throughout the chapter, we will use two different models for the arrivals and departures of elements: a discrete finite model with a finite number of elements; and a fluid model based on differential equations with an infinite number of elements. Our objective is to model a constant load, i.e. a constant number of elements in the system, so that departing elements are replaced by arriving elements.

In both of the models we start at time \(t = 0\) with all the \(n\) elements placed in the overflow list. The description of the differences between the models is given below.
Discrete Finite Model — In the discrete finite model, we assume that time is divided into time-slots of unit duration. At the start of each time-slot $t > 0$, an element is chosen uniformly at random among all $n$ elements in the system to depart. Next, at the end of time-slot $t$, a new element arrives and is inserted according to the hashing scheme into either a non-full bucket or the overflow list. Therefore, by the end of each time-slot $t$, there are always $n$ elements in the system, either in the hash table or in the overflow list.

Fluid Model — The second model is the fluid model, which attempts to model the behavior of the continuous system as both the number of elements $n$ and the number of buckets $m$ go to infinity with a constant limit ratio $ch = \lim_{n \to \infty} \frac{n}{m}$. In the fluid model, we will often analyze the system using differential equations, and will be mainly interested in their fixed-point solutions.

In this model, each element stays in the system for an exponentially-distributed duration of average $1$. Therefore, at each infinitesimal time-interval $[t, t + \delta t]$, the probability that a given element departs is $n \cdot \delta t + o(\delta t)$. As such, the departure rate from each bucket is proportional to the bucket size.

For each element departure, another element is automatically generated and inserted in the system. Thus, the average arrival rate per bucket is $\frac{n}{m}$, since the arrival rate is $n$ and there are $m$ buckets. Therefore, in the fluid model, we model a constant average arrival rate per bucket of $ch = \lim_{n \to \infty} \frac{n}{m}$.

Consider a finite number of buckets. When arriving elements use a uniformly-distributed hash function, they hash into each bucket at a rate equal to the average rate of $\frac{n}{m}$. However, since we consider in the fluid model an infinite number of buckets, the uniformly-distributed hash function translates to a continuous uniformly-distributed hash function. By extension, and for simplicity, we will define such a function as one that enables the same arrival rate of $ch$ to all buckets.

Furthermore, we will define the average number of memory accesses per element $a$ such that it is valid at any time $t$, thus we call it the hashing rate. We will also assume that the hash values are independent from the bucket occupancy, i.e. that the hashing rate of $a$ is valid given any bucket size.

Model Alternatives — In general, to model system scaling, we would be interested in using the discrete finite model, and then in studying how its solution scales with $n$. However, given the complex interactions between the $n$ elements, this model often prove intractable. Therefore, we will use the fluid model in these cases, and will most often not be able to prove convergence of the discrete finite model to the fluid model. Likewise, we will not always prove convergence of the differential equations to the fixed-point solutions. This is, of course, a limit of our analysis.

On the other hand, for the single-choice hashing scheme (Section 5.4), we provide a full analysis with both models, and prove that the limit of the discrete finite model behaves indeed like in the fluid model. In simulations, we will also show that the scaled systems converge fast to their fluid model. We refer to [96] for a more complete discussion of the sufficient conditions for the convergence to the fluid-limit fixed-point solution.

57
5.4 A Single-Choice Hashing Scheme

We start by analyzing a simplistic hashing scheme, which uses only a single uniformly-distributed hash function \( H \) to insert elements in the hash table. Each element \( x \) is stored in bucket \( H(x) \), if it is not full, and in the overflow list otherwise. Since an element uses exactly one hash function, its average number of memory accesses per element is \( a = 1 \). Of course, this simplistic scheme would probably not be implemented in advanced networking devices. However, it provides a better intuition on the reasons behind the performance degradation in dynamic hash-table schemes.

**Discrete Finite Model** — We first develop an analytical model for the scheme within the discrete framework presented in Section 5.3. Let \( p_k(t) \) denote the expected fraction of buckets that have \( k \) elements at the end of time-slot \( t \), and \( p(t) = (p_0(t), p_1(t), \ldots, p_h(t)) \). Using the discrete finite model, we obtain the following result on the limits of \( p(t) \) and of the expected overflow fraction. The full proof is based on a birth-death Markov chain that models the occupancy of an arbitrary bucket over time.

**Theorem 8** Let \( C = \sum_{\ell=0}^{h} \binom{n}{\ell} \left(\frac{1}{m-1}\right)^{\ell} \). In the discrete finite model, when \( t \to \infty \),
(i) \( p(t) \) converges to the Engset distribution \( \pi^n \) \cite{100, 101}; that is, for all \( 0 \leq k \leq h \), \( p_k(t) \) converges to \( \pi^n_k \), where
\[
\pi^n_k = \frac{1}{C} \cdot \binom{n}{k} \cdot \left(\frac{1}{m-1}\right)^{k}. \tag{5.1}
\]
(ii) the expected overflow fraction converges to
\[
\frac{1}{C} \cdot \binom{n}{h} \cdot \left(\frac{1}{m-1}\right)^{h} \cdot \left(1 - \frac{h}{n}\right). \tag{5.2}
\]

**Proof.** We model the hash table using a discrete-time Markov chain that represents the occupancy \( X^n_i \) of an arbitrary bucket \( i \) at the end of time-slot \( t \). We will see that this is possible because the process is memoryless from time-slot to time-slot, and because when conditioned on the occupancy of bucket \( i \), its arrival and departure probabilities are independent of the states of the other buckets or of the overflow list.

At the end of each time-slot \( t - 1 \), there are \( X^n_{i-1} \) elements in bucket \( i \). Then, at the start of time-slot \( t \), the element that departs is chosen uniformly at random out of the \( n \) elements in the system. Therefore, the probability that it belongs to one of the \( X^n_{i-1} \) elements in bucket \( i \) is \( \frac{X^n_{i-1}}{n} \).

The element is then reinserted into the system. The probability that it is hashed by the uniformly-distributed hash function \( H \) into bucket \( i \) out of \( m \) buckets is \( \frac{1}{m} \).

We can now build the state transition matrix. The bucket occupancy obviously increases iff there is no departure while there is an arrival, while it decreases iff there is a departure but no arrival. For \( 1 \leq j, k \leq h \), the transition probability from occupancy \( j \) to occupancy
The birth-death Markov chain is clearly irreducible, positive, recurrent and aperiodic. Therefore, it converges to its stationary distribution $\pi^n$. In addition, since the state transition matrix does not depend on $i$, by ergodicity, $p(t)$ also converges to $\pi^n$, as detailed in the theorem.

Lastly, bucket $i$ overflows in a given time-slot $t$ when it contains $h$ elements, no element leaves bucket $i$, and an element arrives to bucket $i$. In addition, the probability that the element arriving at time $t$ is sent to the overflow list is the sum of all individual bucket overflow probabilities. Therefore, by ergodicity, the total overflow probability at time-slot $t$ converges to

$$
\gamma^n_{\text{SINGLE}} = m \cdot \left( \pi^n_h \cdot \left( 1 - \frac{h}{n} \right) \cdot \frac{1}{m} \right) = \pi^n_h \cdot \left( 1 - \frac{h}{n} \right).
$$

Since the mean time for which an element stays in the overflow list is $n$, we immediately get by Little’s Law that the expected number of elements in the overflow list is $(\gamma^n_{\text{SINGLE}} \cdot n)$.

Equation (5.1) can be rewritten as a truncated binomial expression

$$
\pi^n_k = \frac{\binom{n}{k} \left( \frac{1}{m} \right)^k \left( 1 - \frac{1}{m} \right)^{n-k}}{\sum_{i=0}^{h} \binom{n}{i} \left( \frac{1}{m} \right)^i \left( 1 - \frac{1}{m} \right)^{n-i}}.
$$

Since the proof of Theorem 8 shows that the distribution of a bucket occupancy also follows $\pi^n_k$ (as $p(t)$ does), the above expression hints at the following interesting equivalent system: the bucket occupancy is distributed as if the $n$ elements were assigned uniformly at random among the $m$ buckets, and then the buckets with more than $h$ elements were completely cleared out and had all their elements put in the overflow list. This is in contrast with the static system in which only elements exceeding the bucket capacity of $h$ are placed in the overflow list. Therefore, Equation (5.3) nicely illustrates the difference between the static and dynamic cases.

We now show a detailed example which shows a simplistic setting where, as the number of buckets increases (with fixed load), the dynamic case yields an expected overflow fraction.
of 50%, while the static case has an expected overflow fraction of only $e^{-1} \approx 36.79\%$.

Example 7 For the case where $h = 1$ the expected overflow fraction $\gamma_{\text{single}}^n$ reduces to $\gamma_{\text{single}}^n = \frac{n-1}{m+n-1}$. Denoting the load $c = \frac{n}{m}$, $\gamma_{\text{single}}^n = \frac{c-\frac{1}{2}}{1+c-\frac{1}{2}} \xrightarrow{m \to \infty} \gamma_{\text{single}} = \frac{c}{1+c}$, where $\gamma_{\text{single}}$ is the limit expected overflow fraction as we scale the system while keeping the load constant to $c$. For instance, for $c = 1$, we get

$$\gamma_{\text{single}} = 50\%.$$ (5.4)

In other words, when scaling the system with the same number of elements and buckets, we find that we asymptotically lose 50% of the elements.

Note that in such a scaling, we lose a fraction $\gamma_{\text{single}} = \frac{1-\frac{1}{2}}{1+c} = \frac{m-1}{2m-1}$ of the elements.

This fraction corresponds for instance to no losses with $m = 1$; to $1/3$ of the elements lost with $m = 2$; and to 40% of the elements lost with $m = 3$; the overflow fraction then continuing to increase monotonically and converge to $\gamma_{\text{single}}$.

Now we compare the dynamic overflow fraction $\gamma_{\text{single}}^n$ with the static overflow fraction, denoted $\sigma_{\text{single}}^n$, given a bucket size of 1. First, assuming a load $c = 1$, the overflow fraction is equal to the fraction of unused buckets, because the number of elements is equal to the number of buckets, and buckets can contain at most one element. Therefore, since each element chooses a bucket uniformly at random, we get

$$\sigma_{\text{single}}^n = \left(1 - \frac{1}{m}\right)^m \xrightarrow{m \to \infty} \sigma_{\text{single}} = e^{-1}.$$ (5.4)

Thus, the static system has a clearly lower overflow fraction.

Fig. 5.3 illustrates the overflow fraction as a function of $m$ in both the static and dynamic cases. Clearly, the dynamic case always causes a higher overflow fraction. In addition, both converge fast from below to their limit values.

More generally, for any arbitrary load $c \leq 1$, the limit static overflow fraction is known [8] to be $\sigma_{\text{single}} = 1 - \frac{1-e^{-c}}{c}$. Therefore, as $c \to 0$, we asymptotically get $\gamma_{\text{single}} = c + O(c^2)$, and $\sigma_{\text{single}} = \frac{c}{2} + O(c^2)$, so for low loads, the large dynamic system has about twice the overflow of the large static one.

60
**Fluid Model** — We now analyze the infinite system using a fluid model. In the fluid model, elements stay in the system for an exponentially-distributed duration of average 1, and therefore the departure rate from each bucket is proportional to the bucket size. In addition, when an element departs, a new element is inserted into the hash table (or in the overflow list if the corresponding bucket is full). As explained in Section 5.3, the arrival rate to each bucket is therefore $c_h = \lim_{n \to \infty} \frac{n}{m}$.

The following theorem, which is based on the $M/M/h/h$ continuous-time Markov process [101], shows the performance of the scheme under the fluid model.

**Theorem 9** In the fluid model, 
(i) $p(t)$ converges to the stationary distribution $\pi^\infty$, where

$$\pi^\infty_k = \frac{(ch)^k}{k!} \left( \sum_{l=0}^{h} \frac{(ch)^l}{l!} \right), \quad k = 0, \ldots, h.$$  

(ii) the expected overflow fraction converges to $\pi^\infty_h$ and follows the Erlang-B formula.

**Proof.** In the fluid model, the departures from buckets of size $k \geq 1$ cause the fraction $p_k(t)$ to decrease at rate $k \cdot p_k(t)$, since each of the $k$ elements departs at rate 1, and therefore the $k$ elements depart at a total rate of $k$. Since the buckets of size $k$ with a departing element have a new size $k - 1$, the departures from such buckets increase in turn $p_{k-1}(t)$ at the same rate $k \cdot p_k(t)$.

Likewise, the arrivals to buckets of size $k < h$, which occur at rate $c_h = \lim_{n \to \infty} \frac{n}{m}$, cause the fraction $p_k(t)$ to decrease at rate $c_h \cdot p_k(t)$, and the fraction $p_{k+1}(t)$ to increase at the same rate.

Therefore, we obtain the following differential equation, which characterizes the birth-death process:

$$\frac{dp_k(t)}{dt} = \begin{cases} 
ch \cdot p_{k-1}(t) + (k + 1)p_{k+1}(t) \\
-(ch + k)p_k(t) & \text{for } k \in [1, h - 1], \\
p_1(t) - ch \cdot p_0(t) & \text{for } k = 0, \\
ch \cdot p_{h-1}(t) - hp_h(t) & \text{for } k = h,
\end{cases}$$

with $\sum_{k=0}^{h} p_k = 1$. Assume the system is initially empty, i.e. $p_k(0) = 1_{k=0}$.

Solving the differential equation above yields the stationary distribution of an $M/M/h/h$ loss system [101]. In addition, the drop rate in the fluid model is

$$\gamma^\infty_{\text{sINGLE}} = \pi^\infty_h,$$

following the well-known Erlang-B formula. Finally, since the differential equations are exactly those of the ergodic $M/M/h/h$ continuous-time Markov process, $p(t)$ converges to $\pi^\infty$ [101, 102].

61
We have seen that the discrete finite model with \( n \) elements yield a stationary distribution \( \pi^n \), while the fluid model yields the distribution \( \pi^\infty \) (from the fixed-point equations). We will now show that as expected, when scaling \( n \) to infinity, \( \pi^n \) converges to \( \pi^\infty \), and so does the associated overflow fraction.

**Corollary 10** When \( n \to \infty \) with \( \frac{n}{m} \to ch \),

(i) the stationary distribution converges to the fixed-point distribution of the fluid model: \( \pi^n \to \pi^\infty \); and

(ii) the expected overflow fraction of the discrete finite model converges to the overflow fraction of the fluid model.

**Proof.** For each \( n \in \mathbb{N}^* \cup \{\infty\} \), \( \sum_{k=0}^{h} \pi^n_k = 1 \) and \( \pi^n_0 > 0 \), so for \( k \in [0, h] \), \( \frac{\pi^n_k}{\pi^n_0} \) is defined and

\[
\pi^n_k = \frac{\sum_{l=0}^{h} \pi^n_l}{\pi^n_0}.
\]

Therefore, to prove the convergence of \( \{\pi^n\}_{n \geq 1} \), which is a sequence of finite vectors, we only need to prove the point convergence of \( \frac{\pi^n_k}{\pi^n_0} \) to \( \frac{\pi^\infty_k}{\pi^\infty_0} \). We get

\[
\frac{\pi^n_k}{\pi^n_0} = \binom{n}{k} \frac{1}{m-1} = \frac{1}{k!} \left( \frac{n}{m} \right)^k \cdot \left( 1 - \frac{1}{n} \right) \cdots \cdot \left( 1 - \frac{k-1}{n} \right)
\]

\[
= \frac{1}{k!} \cdot (ch)^k \cdot (1 + o(1)) = \frac{\pi^\infty_k}{\pi^\infty_0} \cdot (1 + o(1)),
\]

which concludes the proof of the convergence of \( \pi^n \) to \( \pi^\infty \).

Lastly, \( \gamma^\infty_{\text{single}} = \pi^\infty_h \) and \( \gamma^n_{\text{single}} = \pi^n_h \cdot (1 - \frac{h}{n}) \). Since \( (1 - \frac{h}{n}) = 1 + o(1) \), the convergence of \( \gamma^n_{\text{single}} \) to \( \gamma^\infty_{\text{single}} \) follows. \( \square \)

Finally, we generalize the scheme to deal with probabilistic insertions. Namely, there exists some \( \alpha \in [0, 1] \) such that each arriving element is either hashed into a bucket as before with probability \( \alpha \), or placed directly in the overflow list with probability \( 1 - \alpha \), yielding an average number of memory accesses \( \alpha \) (or equivalently, a total number of memory accesses \( \alpha n \leq n \), less than the number of elements). Using the fluid model for simplicity, we obtain the following result. While this probabilistic scheme is probably not useful in practice (since the average memory access rate is seldom less than \( 1 \)), we will later demonstrate that it is optimal under specific conditions.

**Theorem 11** In the fluid model, given the single-choice hashing scheme with an insertion probability \( \alpha \), we obtain \( a = \alpha \leq 1 \), and
(i) $p(t)$ converges to the stationary distribution $\pi^\infty$, where

$$
\pi^\infty_k = \frac{(\alpha ch)^k}{k!} / \sum_{\ell=0}^{h} \frac{(\alpha ch)^\ell}{\ell!}, \quad k = 0, \ldots, h.
$$

(ii) the expected overflow fraction converges to $(1 - \alpha) + \alpha \cdot \pi^\infty_h$.

**Proof.** The differential equations are the same as in the proof of Theorem 9 when replacing $ch$ by $\alpha ch$, since $\alpha$ simply changes the arrival rate. The distribution results are then immediate. In addition, in the fixed-point equations, an arriving element either overflows immediately with probability $1 - \alpha$, or checks with probability $\alpha$ a bucket that can be full with probability $\pi^\infty_h$, hence the overflow equation follows as well. $\square$

### 5.5 Overflow Lower Bound

Our objective is to find a lower bound on the expected overflow fraction $\gamma$ of any $(a, d, c, h)$ hashing scheme, when assuming a fluid model. We will study the simpler case with uniformly-distributed hash functions, as defined in Section 5.3. The more general case with several hash functions using different subtable-based distributions appears in Section 5.6.

The proof relies on the following result from [103]. Consider an Erlang blocking model with $N$ servers, and suppose that the arrival rate depends on the system. Let $X(t)$ by the number of transmissions in progress at time $t$, and $\lambda_k$ be the arrival rate when there are $k$ transmissions in progress, $k = 0, 1, \ldots, N - 1$. Then we have:

**Lemma 1 (Theorem 4.2 in [103])** For all increasing mappings $f : \mathbb{R} \to \mathbb{R}$ and for all $t > 0$, $Ef(X(t))$ is concave increasing as a function of $\lambda_k$, for $k = 0, 1, \ldots, N - 1$.

We use this lemma to prove the lower-bound result.

**Theorem 12** In the fluid model, under the assumptions above where only uniformly-distributed hash functions are used (see Definition 1), the optimal expected fixed-point overflow fraction $\gamma_{\text{opt}}$ in the OPTIMAL DYNAMIC HASH TABLE PROBLEM is lower-bounded by

$$
\gamma_{\text{lb}}(a) = 1 - a + a \cdot \frac{r^h}{h!} / \sum_{l=0}^{h} \frac{r^l}{l!},
$$

where $r = ach$.

**Proof.** For any $k \in [1, h]$, let $p_k(t)$ denote the fraction of buckets of size $k$. As shown with the single-choice hashing scheme (SINGLE), in the fluid model, the departures from buckets of size $k \geq 1$ decrease the fraction $p_k(t)$ at rate $k \cdot p_k(t)$, and increase the fraction $p_{k-1}(t)$ at the same rate $k \cdot p_k(t)$.
Likewise, the element arrival rate before hashing is $c_h = \lim_{n \to \infty} \frac{n}{m}$, and the hashing rate per element is $a$, therefore the hashing rate is $a c_h$. Let $r = a c_h$. Since elements can only decide to enter a bucket after hashing into it, we know that their post-hashing arrival rate to any bucket is bounded from above by $r$. Of course, the decision of whether to enter a bucket after hashing into it might depend on the bucket occupancy. Therefore, let $r_k(t)$ denote the average arrival rate to the fraction of buckets of size $k$ at time $t$, with $0 \leq r_k(t) \leq r$. These arrivals cause the fraction $p_k(t)$ to decrease at rate $r_k(t) \cdot p_k(t)$, and the fraction $p_{k+1}(t)$ to increase at the same rate.

Combining departures and arrivals, we obtain the following differential equation characterizing the birth-death process:

$$\frac{dp_k(t)}{dt} = \begin{cases} r_{k-1}(t)p_{k-1}(t) + (k+1)p_{k+1}(t) & \text{for } k \in [1, h-1], \\ -(r_k(t) + k)p_k(t) & \text{for } k = 0, \\ p_1(t) - r_0(t)p_0(t) & \text{for } k = h, \\ r_{h-1}(t)p_{h-1}(t) - hp_h(t) & \text{for } k = h, \end{cases}$$

with $\sum_{k=0}^{h} p_k = 1$ and $p_k(0) = 1_{k=0}$.

Consider a fixed point $\pi$ of the birth-death process, i.e. assume that for any $i \in [0, h]$, $\frac{d\pi_i(t)}{dt} = 0$. Then the finite vector $(\pi_0(t), \ldots, \pi_h(t))$ is independent of $t$, and therefore the arrival rate $r_k(t)$ to a bucket of size $k$ is independent of $t$ as well. Denote this constant arrival rate as $r_k$, where $0 \leq r_k \leq r$. Solving the differential equation above yields the following balance equations: for each $k \in [1, h]$,

$$r_{k-1}\pi_{k-1} = k\pi_k,$$

with $\sum_{k=0}^{h} \pi_k = 1$. Therefore, $\pi$ satisfies

$$\pi_k = \frac{\prod_{j=0}^{k-1} r_j}{k!} \cdot \pi_0 = \frac{\prod_{j=0}^{k-1} r_j}{k!} \cdot \frac{\prod_{l=0}^{h-1} \prod_{j=0}^{l-1} r_j}{l!}.$$ 

Using Lemma 1, we find that the average bucket occupancy $\mathbb{E}X^\pi$ under $\pi$ is upper-bounded by the average bucket occupancy $\mathbb{E}X^\pi$ under $\bar{\pi}$, where $\bar{\pi}$ is the fixed-point distribution when $r_k = r$. This is because $f : X \to X$ is increasing [102], and $r_k \leq r$ for $k \in [0, h-1]$. Therefore,

$$\mathbb{E}X^\pi \leq \mathbb{E}X^\bar{\pi}.\quad (5.9)$$

Finally, note that without element losses, we would have the average occupancy equal to the average number of elements per bucket, i.e., $c_h = \lim_{n \to \infty} \frac{n}{m}$. Therefore, the average
fraction of lost elements is equal to

\[ \gamma^\infty = \frac{c h - \mathbb{E}X^n}{ch} \equiv 1 - a \cdot \frac{\mathbb{E}X^n}{r} \geq 1 - a \cdot \frac{\mathbb{E}X^n}{r} \]

\[ = 1 - a \cdot \frac{r \cdot (1 - \pi_h)}{r} \]

\[ = 1 - a + a \cdot \frac{c h}{c h} \]

\[ = 1 - a + a \cdot \frac{r!}{\sum_{l=0}^{h} (r l!)} \]

where (a) uses \( r = ach \), (b) relies on Equation (5.9), (c) uses a standard Erlang-B result [100, 101], and (d) comes from Equation (5.9) with \( r_k = r \). \( \square \)

Note again that the Erlang-B formula appears in the lower-bound on the overflow. This yields the following optimality result:

**Theorem 13** In the fluid model, the single-choice hashing scheme is optimal for every average number of memory accesses \( a \) in \([0, 1]\) (and in particular for \( a = 1 \)).

**Proof.** For the single scheme, there is a single hashed bucket per element, and it is accessed with probability \( \alpha \), therefore \( a = \alpha \). For \( a \leq 1 \), we get

\[ \gamma_{\text{LB}}^\infty (a) \equiv (1 - a) + a \cdot \frac{(ach)^h}{h!} \sum_{l=0}^{h} \frac{(ach)^l}{l!} \equiv \gamma_{\text{SINGLE}}^\infty \]

where (a) comes from Equation (5.7), \( r = ach \) and \( a = \alpha \), and (b) from Theorem 11. \( \square \)

**Example 8** We illustrate the significance of the lower bound by considering a simple system with buckets of size \( h = 1 \), implying

\[ \gamma_{\text{LB}}^\infty (a) = 1 - a + a \cdot \frac{c \cdot a}{1 + c \cdot a} = 1 - \frac{a}{1 + c \cdot a} \]

In particular, for a load \( c = 1 \), corresponding to the scaling case where the number of buckets is kept equal to the number of elements and therefore \( \lim_{n \to \infty} \frac{n}{m} = 1 \), we get

\[ \gamma_{\text{LB}}^\infty (a) = 1 - \frac{a}{1 + a} = \frac{1}{1 + a} \]

which shows that the lower-bound decreases slowly as \( \Theta(1/a) \) when the average number of memory accesses per insertion \( a \) increases.

For instance, to get a 1% drop rate we need each element to access an average of at least \( a = 99 \) buckets. Of course, this is impossible to implement in high-speed networking devices. Thus, this lower bound is essentially an impossibility result, which shows that it is not easy to obtain efficient hash tables with deletions.

Fig. 5.4 compares this drop rate lower-bound with the drop rate lower-bound in the...
Figure 5.4: Expected overflow fraction as a function of the average memory access rate $a$.

static case, which is equal to $e^{-a}$ [8]. As $a$ increases, the figure shows how dynamic hash tables are significantly less efficient than their static counterparts.

5.6 Lower Bound with Multiple Hash-Function Distributions

We now consider a setting with a set $\mathcal{I}$ of $I = |\mathcal{I}|$ subtables, where subtable $i \in \mathcal{I}$ uses a fraction $\alpha^i$ of all buckets. We will allow for the $d$ hash functions to use up to $d$ different distributions $\{f_j\}_{1 \leq j \leq d}$ over the $I$ subtables, where each distribution $f_j$ assigns a probability $f^j_i$ to subtable $i \in \mathcal{I}$, with $\sum_{i \in \mathcal{I}} f^j_i = 1$, and then uniformly picks buckets within each subtable (as defined in Section 5.3). We also assume that each distribution $f_j$ is used by a fraction $\kappa_j$ of the total memory accesses. Therefore, subtable $i$ is accessed with a total probability of $\beta^i = \sum_{j=1}^d \kappa_j \cdot f^j_i$, with $\sum_{i \in \mathcal{I}} \beta^i = 1$. The following result establishes that the lower-bound is reached when the hash table is used in a uniform way, i.e. the probability $\beta^i$ of accessing a subtable is equal to its fraction $\alpha^i$ in the table, and therefore the lower-bound is the same as established previously in Theorem 12.

**Theorem 14** In the fluid model with multiple distributions as defined above, the lower-bound $\gamma^\infty_{lb}(a)$ on the fixed-point overflow fraction is the same as the one with a unique uniform hash function, and is reached iff for all $i \in [1, I]$, $\beta^i = \alpha^i$, i.e. the weighted average of all distributions is uniform.

**Proof.** As in the proof of Theorem 12, in each subtable $i$, we focus on the fixed-point distribution $\pi^i$, which satisfies

$$\pi^i_k = \frac{\prod_{j=0}^{k-1} r^i_j}{k!} / \left( \sum_{l=0}^{h} \frac{\prod_{j=0}^{l-1} r^i_j}{l!} \right),$$

with $r^i_j \leq \frac{\beta^i}{\alpha^i} \cdot r$. This is because the rate at which the elements check a bucket in subtable $i \in \mathcal{I}$ is proportional to the ratio of their probability $\beta^i$ of picking subtable $i$ by the proportional size $\alpha^i$ of subtable $i$. In addition, even if the elements check a bucket of size
they can decide not to enter it. The rate \( r_j \), at which they enter it, depends both on the size \( j \) and on the subtable \( i \), although it needs to be upper-bounded by the rate \( \frac{\beta_i}{\alpha^i} \cdot r \) at which they checked it.

From the proof of Theorem 12, in each subtable \( i \in \mathcal{I} \), we know that the average occupancy is upper-bounded by the case in which we have equality

\[
r_j = \frac{\beta_i}{\alpha^i} \cdot r.
\]

We now want to find the vector \( \beta \) that maximizes the average occupancy of the whole system. Let \( \pi \left( \frac{\beta_i}{\alpha^i} \right) \) denote the distribution that maximizes the average occupancy in subtable \( i \in \mathcal{I} \), and define \( f : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \) with \( f(x) = \mathbb{E}X^\pi(x) \). Then we want to find

\[
\max \sum_{i \in \mathcal{I}} \alpha^i \cdot f \left( \frac{\beta_i}{\alpha^i} \right) \quad \text{s.t.} \quad \sum_{i \in \mathcal{I}} \alpha^i = 1, \quad \sum_{i \in \mathcal{I}} \beta^i = 1.
\]

\( f \) is known to be strictly concave [104, 105, 106] (the concavity also follows from Lemma 1). Therefore

\[
\sum_{i \in \mathcal{I}} \alpha^i \cdot f \left( \frac{\beta_i}{\alpha^i} \right) \leq f \left( \sum_{i \in \mathcal{I}} \alpha^i \cdot \frac{\beta_i}{\alpha^i} \right) = f \left( \left( r \cdot \sum_{i \in \mathcal{I}} \beta_i \right) \right) = f(r),
\]

where (a) uses concavity and \( \sum_{i \in \mathcal{I}} \alpha^i = 1 \), and (b) uses \( \sum_{i \in \mathcal{I}} \beta^i = 1 \), with equality iff \( \frac{\beta_i}{\alpha^i} r \) independent of \( i \), i.e. \( \beta^i = \alpha^i \) for all \( i \in \mathcal{I} \), because \( \sum_{i \in \mathcal{I}} \beta^i = \sum_{i \in \mathcal{I}} \alpha^i \). Finally, as in the proof of Theorem 12, the same upper bound on the average bucket occupancy corresponds to the same lower bound on the overflow fraction.

\[\Box\]

### 5.7 A Multiple-Choice Hashing Scheme

We now introduce a natural extension to the single-choice hashing scheme that uses an ordered set of \( d \) hash functions \( \mathcal{H} = \{H_1, \ldots, H_d\} \), such that all the hash functions are independent and uniformly-distributed. Upon inserting an element \( x \), the scheme successively reads the buckets \( H_1(x), \ldots, H_d(x) \) and places \( x \) in the first non-full bucket. If all these buckets are full, \( x \) is placed in the overflow list. To keep an average number of memory accesses per element of at most \( a \), the algorithm attempts to insert \( x \) into the hash table with a probability \( \alpha \), otherwise it is directly placed in the overflow list. We next show an example of this scheme.

**Example 9** Fig. 5.5 illustrates the multiple-choice hashing scheme (MULTIPLE) with \( m = 12 \), \( h = 1 \), \( d = 2 \) and \( q = 1 \). We can see that element \( x_1 \) is initially mapped by \( H_1 \) to a full bucket. It is therefore mapped again by \( H_2 \), and inserted in an empty bucket. Then, the element in bucket number 6 is deleted. On the next step, element \( x_2 \) is directly inserted in
an empty bucket, and therefore does not need a second memory access.

We evaluate the performance of this scheme analytically using the fluid model.

**Theorem 15** Assume the multiple-choice hashing scheme with a hashing probability \( \alpha \). Using the fluid-model fixed-point distribution \( \pi^\infty \),

(i) \( \pi^\infty \) satisfies \( \pi_k^\infty (a) = \frac{(ach)^k}{k!} \sum_{l=0}^{h} \frac{(ach)^l}{l!} \), for each \( k = 0, \ldots, h \);

(ii) the average bucket access rate \( a \) satisfies the fixed-point equation \( a = \alpha \cdot \frac{1-\pi_h^\infty (a)^d}{1-\pi_h^\infty (a)} \);

(iii) the expected overflow fraction is equal to the lower-bound, and is therefore optimal, for \( a \in [0, a^\alpha] \), where \( a^\alpha \) satisfies the fixed-point equation \( a^\alpha = \frac{1-\pi_h^\infty (a^\alpha)^d}{1-\pi_h^\infty (a^\alpha)} \).

**Proof.** For a given rate \( a \), the differential equations are the same as for the single-choice hashing scheme (SINGLE) and satisfy the same fixed-point distribution (Theorem 13).

Let us now compute \( a \). Following the definition of the fluid model, there is an independence in the following sense: Whenever an element arrives, the probability that it uses its \( l \)th hash function \( H_l \), for \( 1 \leq l \leq d \), is \( \alpha \cdot (\pi_h^\infty)^{l-1} \); namely, the product of the probability \( \alpha \) that it is not directly placed in the overflow list by the probability that the first \( l - 1 \) hash functions mapped into full buckets. Then, the \( l \)th trial is successful with probability \( 1 - \pi_h^\infty \). Finally, there were \( d \) unsuccessful trials with probability \( \alpha \cdot (\pi_h^\infty)^{d-1} \). Therefore, the average number of trials per element is:

\[
a = \left( \sum_{l=1}^{d-1} l \cdot \alpha \cdot (\pi_h^\infty)^{l-1} (1 - \pi_h^\infty) \right) + d \cdot \alpha \cdot (\pi_h^\infty)^{d-1}
\]
Using the general formula
\[
\sum_{k=1}^{K} kx^{k-1} = \frac{1 - x^{K+1} - (1 - x) \cdot (K + 1)x^{K}}{(1 - x)^2},
\]
we get
\[
a = \alpha \left[ \frac{1 - (\pi^\infty)^d - (1 - \pi^\infty) \cdot d \cdot (\pi^\infty)^{d-1}}{1 - \pi^\infty} + d \cdot (\pi^\infty)^{d-1} \right]
\]
\[
= \alpha \cdot \frac{1 - (\pi^\infty)^d}{1 - \pi^\infty}.
\]
Finally, this can only hold for \( \alpha \leq 1 \); once we reach \( \alpha = 1 \), we obtain \( a^{\text{multiple}} \).

The following example illustrates our results.

Example 10 For the case where \( h = 1 \), solving the fixed-point equation yields \( a^{\text{co}} = \frac{2c - 1 + \sqrt{1 + 4c^2}}{2c} \). Therefore, for a load of one element per bucket, i.e. \( c = \lim_{n \to \infty} \frac{a}{m} = 1 \), we get \( a^{\text{co}} = \frac{1 + \sqrt{5}}{2} \approx 1.62 \), and the corresponding expected overflow fraction is \( \gamma^{\infty}_{\text{lb}} (a^{\text{co}}) = 1.5 - \frac{\sqrt{5}}{2} \approx 38.2\% \). Likewise, for a load of \( c = 0.1 \), we get \( a^{\text{co}} = \frac{-0.8 + \sqrt{1 + 0.04}}{0.2} \approx 1.099 \), with the corresponding expected overflow fraction \( \gamma^{\infty}_{\text{lb}} (a^{\text{co}}) \approx 0.98\% \).

5.8 Moving Back Elements

So far, we have found optimal schemes for a range of values of \( a \), the average number of memory accesses per element. However, although optimal, the expected overflow fraction may still be too large.

In the literature, several solutions exist to reduce the drop rate (or collision probability) in a dynamic system. One such solution uses limited hash functions in order to be able to rebalance the hash table in case of deletion [18]. However, this approach gives up randomness, and the efficiency of a similar approach appears limited [28]. Another solution, based on the second-chance scheme [47], moves elements from one bucket to another by storing hints at each bucket [28]. These hints help to find another element stored in another bucket that can be moved upon the deletion. However, we found in simulations that this solution was less effective than our suggested scheme presented below for higher loads, while it was more effective for lower loads. Detailed simulation results are found in Section 5.9.

To reduce the overflow fraction, we suggest a scheme that allows moving elements back from the overflow list to the buckets upon a deletion operation³. This scheme can be combined with any insertion scheme.

³We also considered a scheme that works upon insertion. Since we found that moving back elements upon deletion performs better in general, schemes that move elements upon insertion are not presented.
5.8.1 Description

Our scheme, called the moving-back scheme (M-B), relies on a (binary) CAM. In general, a CAM stores keys in entries. Given some key $k$, a parallel lookup is performed over all entries and the index of the first (that is, highest priority) entry that contains $k$ is returned from the CAM. In many cases, this index is later used in order to access in regular memory a direct-access array that contains the value associated with $k$. CAMs enable constant-time operations, however they are more expensive and consume more power than regular memory. It is a common practice to implement the overflow list in a CAM [47, 8, 22], relying on the fact that the number of elements in the overflow list is small.

Our scheme uses an auxiliary CAM, besides the primary CAM used to store the element of the overflow list: For each element $x$ that is stored in the $i$-th entry of the primary CAM, we store the values $\{H_1(x), H_2(x), \ldots, H_d(x)\}$ in entries $d \cdot i, d \cdot i + 1, \ldots, d \cdot i + (d - 1)$ of the auxiliary CAM.

When an element is deleted from a bucket $j$ that was previously full, we need to move an element $x$ from the overflow list to bucket $j$ such that $j$ is the result of applying at least one of the hash-functions on $x$. We can locate such an element in constant time by querying the auxiliary CAM with key $j$. Suppose the entry returned by the auxiliary CAM is $\ell$, then $x$ is located in entry $\lfloor \ell/d \rfloor$ of the primary CAM.

We note that upon moving an element back to the hash table, one should update the corresponding entries of the primary and auxiliary CAMs. An efficient way to update is to write the value $m + 1$ in these entries, such that when a new element is inserted into the overflow list, one can query the auxiliary CAM with the value $m + 1$ to decide in which entry (of the primary CAM) to put the new element.

5.8.2 Analysis

We first derive the exact expected overflow fraction in the case of the single scheme, and later provide an approximate model for the multiple scheme, which is confirmed by simulations.

**Theorem 16** Consider the single scheme with M-B for moving back elements from the TCAM and a symmetric insertion algorithm. The expected overflow fraction is given by:

$$\gamma_{\text{LB}}(a) = 1 - \frac{1}{c} + \frac{1}{c h} e^{-ac h} \sum_{k=0}^{h} (h - k) \frac{(ach)^k}{k!},$$

**Proof.** Whenever a deletion occurs, the CAM device performs a lookup operation for any element that can be moved back to the bucket. Since every element has only one hash value, all elements that correspond to some bucket can be viewed as its own waiting list. Since the element we choose to delete follows a random process that is independent of any other random process in our system, and also the load is fixed, we conclude that the overflow fraction follows the static case exactly, which is given in [8].
Theorem 17 Consider the multiple scheme with M-B for moving back elements from the CAM and a symmetric insertion algorithm. Let $X_i^t$ be the occupancy of bucket $i$ at step $t$ and $P_0, \ldots, P_h$ be the equilibrium probabilities of the occupancy of each buffer. The probabilities can be modeled by the following Markov chain:

$$P^i_{kj} = \Pr (X_t = j | X_{t-1} = k) = \begin{cases} 
    \frac{g}{m} & j = k + 1, k < h \\
    \frac{k}{n} & j = k - 1, h > k > 0 \\
    \frac{k}{n} \cdot e^{-\gamma chd} & j = k - 1, k = h 
\end{cases}$$

where $g = \sum_{l=1}^{d} P_h^{l-1} = \frac{1-P_h^{-d}}{1-P_h^{-1}}$, and the expected overflow fraction $\gamma$ is given by $\gamma = 1 - \frac{1}{e^{\gamma chd}}$.

Proof. The Markov chain is the same as in the regular multiple scheme, except when an element is deleted from a full bucket. In this case, it is possible that one of the overflow elements in the CAM is moved back to the bucket. It is possible only in this case because all elements in the CAM have hashes to full buckets.

We now approximate the probability that none of the elements has a hash value to that bucket: The total number of hashes is $\gamma \cdot n \cdot d$, where all the hashes are to full buckets. The number of full buckets is $P_h \cdot m$. The probability that a single hash does not point to the specific bucket is $\frac{P_h^{-m}}{P_h^{-m}} = 1 - \frac{1}{P_h^{-m}}$. And the probability that none of them points to the specific bucket is given by

$$\left( 1 - \frac{1}{P_h^{-m}} \right)^{\gamma nd} \approx e^{-\gamma chd} = e^{-\gamma chd}.$$

Multiplying the above expression by the probability that one of the elements is picked for deletions in case the bucket is full yields the claimed Markov chain. \(\square\)

5.9 Experimental Results

5.9.1 Simulations

Fig. 5.6 compares all the schemes by plotting the expected overflow fraction $\gamma$ (Definition 5), as a function of the average number of memory accesses $a$. It was obtained with $d = 4$ choices, bucket size $h = 4$, $n = 4,096$ elements and $m = 1,024$ buckets, yielding a load $c = 1$.

The solid line plots the expected overflow fraction lower-bound $\gamma_{\text{LB}} (a)$ from Theorem 12. Simulations show that the proposed M-B scheme beats the lower bound with an expected overflow fraction of 4.6%, emphasizing the strength of this architecture. Of course, the lower bound does not apply to this case, since it moves back elements from the CAM.

As follows from Theorems 13 and 15, the expected overflow fractions $\gamma_{\text{single}} (a)$ and $\gamma_{\text{multiple}} (a)$ of the single-choice (SINGLE) and the multiple-choice (MULTIPLE) hashing
schemes follow the lower-bound line, respectively until $a_{\text{SINGLE}}^0 = 1$ with $\gamma_{\text{SINGLE}} = 31.1\%$, and $a_{\text{MULTIPLE}}^0 = 2.195$ with $\gamma_{\text{MULTIPLE}} = 13.5\%$. Therefore, they are clearly optimal up to a certain point.

We also test our models from Section 5.8.2. Fig. 5.7 shows the accuracy of our M-B model. We ran simulations with $m = 1024$, $h = 4$, $d = 2$ and different loads. The maximum gap is for load $c = 1$ where our model predicts an expected overflow fraction of 9.20%, whereas simulations show an expected overflow fraction of 9.68%. For lower values of $c$, the model is much more accurate. For instance, for load $c = 0.5$, our model predicts an expected overflow fraction of 0.19% compared to an overflow fraction of 0.18% found via simulations.

We further evaluate the performance of our proposed M-B scheme. Quite surprisingly, when using the MULTIPLE scheme (of Section 5.7), the M-B scheme outperforms the static case of the MULTIPLE scheme (see Fig. 5.8), and performs similarly to the static $d$-random scheme (in the static case, $d$-random performs better than our multiple-choice scheme, albeit consuming significantly more energy [8]). This can be explained intuitively as follows: our moving-back strategy moves back an element to the only corresponding bucket which is not full; this is equivalent to inserting the element to the least occupied bucket as in the

Figure 5.6: Expected overflow fraction as a function of $a$ with $d = 4$, $h = 4$, $c = 1$.

Figure 5.7: M-B with MULTIPLE scheme, for $h = 4$, $d = 2$ and different loads
d-random hashing scheme.

Finally, we compare the performance of our proposed M-B scheme with the performance of the hint-based scheme proposed in [28]. Note that our M-B scheme can be used with any insertion scheme. Thus, for fair comparison, since the hint-based scheme uses the second-chance scheme [47] for insertions, we also used the second-chance scheme for our proposed M-B scheme. We ran simulations with $m = 4096$, $h = 1$, $d = 4$ and different loads. As proposed in [28], the memory level sizes are exponentially decreasing with factor 2.

Fig. 5.10 shows that our M-B scheme is more effective than the hints-based scheme for higher loads, while it is less effective for lower loads. For instance, for a load of 0.6, the M-B and the hints-based schemes yield expected overflow fractions of 1.08% and 0.78%, respectively. For a load of 0.7, they yield 3.88% and 4.59%.

### 5.9.2 Experiments Using Real-Life Traces

We have also conducted experiments using real-life traces recorded on a single direction of an OC192 backbone link [98]. Our goal is to compare the average overflow fraction retrieved using our models for single and multiple with the corresponding overflow fraction when using a real hash function on a real-life trace. We used a 64-bit mix function [99] to implement two 16-bit hash functions. We used $m = 10,000$ buckets, and set a number of
Figure 5.10: The M-B and the hints-based schemes, for \( h = 4, d = 1 \) and different loads.

Figure 5.11: Marginal overflow fraction of 100 on-off flows with \( m = 500, h = 1 \) and \( d = 2 \) elements \( n \) as corresponding to various values of \( h \) and \( c \). To keep a constant desired load, we alternated 100,000 times between an arrival (insertion) of a new TCP packet according to the trace, and the departure (deletion) of a random TCP packet. The hash functions were given the source and destination IP tuple as well as the sequence and acknowledgment numbers of the TCP packets. Therefore, the hash table stores the latest TCP packets, and can retrieve any needed packet based on its header. It can be used to monitor ongoing TCP flows, given a target number \( n \) of packets that are stored at any time. Its objective in our experiments was mainly to test the correctness of our model.

Fig. 5.9 shows that the results of our experiments are relatively close to our model. The maximum gap is for the SINGLE scheme with \( h = 1 \) and \( c = 0.3 \). Our model predicts an average overflow fraction of 23.08\%, while the experiment yields 25.67\%.

5.9.3 Experiments Using an On-off Arrival Model

We also consider a queueing model where at each step \( i \), \( b_i \) elements arrive according to \( k \) independent on-off bursty flows of elements [107]; then, after the arrival phase, one element is randomly deleted. Therefore, the number of elements in the system keeps changing, contrarily to the previous models with a constant load.

Fig. 5.11 shows the marginal overflow fraction under the above queueing model with \( k = 100 \) on-off flows of elements. Each flow has rate \( \rho = 0.0095 \) and average burst size of 10
elements. The figure shows that, given the number of elements currently in the system, the
marginal overflow fraction is approximately the one we found for the constant-load case,
both for single and multiple.

Moreover, by the distribution of the number of elements in the system given by the
queueing model, we are able to heuristically approximate the overall expected number of
elements in the overflow list. More precisely, we take the sum-product of the queue size
distribution by the distribution of the overflow fraction as a function of the load. In the
case of single this model gives an expected number of overflow elements of 61.63, while
simulations yield 61.41. Likewise, for multiple, we obtain 40.17 and 40.26, respectively.
Therefore, this heuristic model proves quite accurate.
Chapter 6

Balancing Table Buckets

In this chapter, we define, analyze and solve a fundamental access-constrained balancing problem, where elements need to be optimally balanced across the buckets while satisfying average and instantaneous constraints on the number of memory accesses associated with checking the current load of the buckets. We then use these results and suggest a new access-efficient Bloom filter scheme in networking devices, called the Balanced Bloom filter.

6.1 Bloom Filter Inefficiency

While the Bloom filter is efficient in memory space, Bloom filters require a significant number of memory accesses. For instance, a Bloom filter with 30 bits per element yields negligible FPR, but also requires to access about $30 \cdot \ln 2 \approx 21$ memory bits per query. Note that each of these bits can reside in an arbitrary location over the memory space. Thus, such a Bloom filter would need a prohibitive memory access bandwidth in networking devices when either implemented in an off-chip setting (that is, requiring to access 21 memory blocks per query) or distributed over a network (equivalently, it may be required to access 21 nodes per query).

One proposal to improve the access-efficiency of Bloom filters is to use a Blocked Bloom filter [29, 30], in which each element is first hashed using a single hash function to one of the memory blocks, and then the memory block operates as a local Bloom filter. Although this technique is clearly access-efficient, since each element requires to access a single memory block, it also suffers from a high FPR, due to a typical load imbalance between the memory blocks. Such an imbalance is inherent in this scheme due to the fact that, given $n$ elements and $n$ memory blocks, the maximum block load is $O \left( \log n / \log \log n \right)$ with high probability, while the average block load is 1 [24].

To tackle this problem, our basic approach is to use load-balancing schemes, making the number of elements in each memory block as balanced as possible. We further propose to use an overflow list that stores elements hashed to overloaded memory-blocks. The overflow list is typically small. For example, it can be implemented using a content-addressable memory
(CAM), which supports parallel-lookup operations.

6.2 Summary of Our Results

In this chapter, we propose a new access-efficient Bloom filter architecture for networking devices, called Balanced Bloom filter. We first maintain balancing schemes to distribute the elements between the memory blocks. At high level, before inserting an element, these schemes query the load of a memory block, and in case it exceeds some threshold the choose another memory block. Memory blocks are chosen by applying hash functions on the inserted element. We also propose to use an optional overflow list to store elements that all hash functions used map to already overloaded memory blocks.

To study this problem, we explore the optimality region of the balancing problem. Namely, we consider different balancing schemes at different loads, and determine several selections of the access budget $a$ and the overflow fraction $\gamma$ such that the balancing scheme is optimal with respect to the cost function $\phi$. In particular, given some access budget $a$ within a predetermined range, we will show that our scheme is optimal for some $\gamma(a)$, and for any $\gamma$ satisfying $\gamma \geq \gamma(a)$, thus defining an optimality region over the $(a, \gamma)$ plane.

To show optimality, we first provide lower bounds on the minimum cost of each instance of the problem. The lower bound depends on the access budget $a$, the number of hash functions $d$, and the overflow list size, but, quite surprisingly, does not depend on the cost function $\phi$. Our lower bounds hold when all hash functions have uniform distribution or when their overall distribution is uniform (in the latter case, the hash function distributions can be different). Then, we provide three different schemes that meet the lower bounds on different access budgets; we further find the minimum size of the overflow list that should be provided in order to achieve optimality. All our analytical models are compared with simulations showing their accuracy.

We conclude by showing how, with a proper choice of the cost function $\phi$, the balancing problem can be directly used to optimize Bloom filters. For example, for an average number of access operations of $a = 1.2$, and $\gamma = 0.5\%$ of the elements stored in the overflow list, the FPR is reduced by up to two orders of magnitude.

Further, since the cost function $\phi$ is general, our approach can have many applications, such as other Bloom filter implementations [31], Counting Bloom filter variants [11, 108], and other applications. We further show how the solution of this problem can be used to construct linked-list–based hash tables with optimal variance.

Finally, we note that concurrently, another work has also proposed to balance the elements in the context of Bloom filters [109]. However, beyond the mere balancing approach, we further provide a theoretical framework for the general problem of balancing the elements subject to a given access budget.

Chapter Organization We first describe our basic architecture of the Balanced Bloom filters in Section 6.3. Then, the optimal balancing problem is defined in Section 6.4, followed
Figure 6.1: Illustration of a Balanced Bloom filter implementation based on MHT-B, with three subtables.

by our lower bound results in Section 6.5. The three optimal schemes and their analysis are presented in Sections 6.6, 6.7, and 6.8, while a comparative study appears in Section 6.10. In Section 6.11 we show how the solution of the balancing problem can be used to construct access-efficient Balanced Bloom filters. Then, we verify our analysis using trace-driven experiments in Section 6.12.

6.3 Balanced Bloom Filters

In this section, we present the basic architecture of Balanced Bloom filter. Our architecture follows the two guidelines of balancing the elements between the memory blocks and the usage of an overflow list.

In our basic architecture, illustrated in Fig. 6.1, each memory block functions as a local Bloom filter, with the only modification that the memory block also saves some bits for a counter storing the number of elements inserted locally.

Although any balancing scheme can be used, for the purpose of the section we rely on a special case of the multi-level hash table (MHT-B) balancing scheme. The memory is divided into $d$ separate subtables $T_1, \ldots, T_d$, with a uniform hash function for each one of these subtables. Upon an element arrival, it is placed in the first subtable in which the corresponding mapped memory block has load lower than a pre-defined threshold $\bar{h}$. If no such memory block exists, the element is placed in the overflow list.

A lookup operation follows the same steps as an insertion operation: The local Bloom
filters of the mapped memory blocks are queried one by one, until either the element is found, or a Bloom filter with load less than $\tilde{h}$ is queried. If all Bloom filters have full load and the element was not found, then the overflow list is also queried.

Fig. 6.1 illustrates an insertion of a new element with $\tilde{h} = 3$. The memory consists of 3 subtals of decreasing size, with 4, 2 and 1 memory blocks, respectively. Each memory block is of size 6 bits, with 2 bits for the counter and 4 for the local Bloom filter. When element $y$ arrives, it is first hashed into the memory block of subtable $T_1$ with address 10. The counter at this memory block indicates that 3 elements have already been inserted. Since this load is equal to the threshold $\tilde{h} = 3$, the scheme then tries to insert the element into subtable $T_2$. In this subtable, the element is hashed into the memory block with address 01, where there are 2 elements. Since $2 < \tilde{h}$, the element is inserted into this memory block. The dashed arrow to subtable $T_3$ illustrates a hash function that is not actually performed. In addition, the element $x$ is in the overflow list because all of its corresponding buckets were full upon its insertion.

### 6.4 Problem Statement

To further study our problem, we first define and solve the optimal access-constrained balancing problem in the following sections. In this section, we define the notations and settings of this balancing problem.

The access-efficiency of a balancing scheme is measured by the number of bucket accesses needed to store the incoming elements. We assume that a balancing scheme needs to access a bucket to obtain any information on it. We do not count accesses to the overflow list.

We further consider two constraints, which can be seen as either power- or throughput-constraints depending on the application. First, we require that the average number of bucket accesses per element insertion must be bounded by some constant $a \geq 0$. In addition, notice that the worst-case number of bucket accesses per element insertion is always bounded by $d$, because an element does not need to consider any of its $d$ hash functions more than once. These two constraints are captured by the following definition:

**Definition 8** An $(a, d, r)$ balancing scheme is a hashing scheme that inserts all elements with an average (respectively, maximum) number of bucket accesses per insertion of at most $a$ (respectively, $d$), when given an element per bucket ratio $r$.

We are now ready to define the optimal balancing problem, which is the focus of this chapter. Let $\phi : \mathbb{N} \mapsto \mathbb{R}$ be the cost function mapping the occupancy of a bucket to its real-valued cost. We assume that $\phi$ is non-decreasing and convex. Our goal is to minimize the expected overall cost:

**Definition 9** Let $O_j$ be a random variable that counts the number of elements in the $j$-th bucket. Given $\gamma$, $a$, $d$ and $r$, the optimal access-constrained balancing problem
consists of finding an \( (a, d, r) \) balancing scheme that minimizes

\[
\phi^{\text{BAL}} = \lim_{m \to \infty} \frac{1}{m} \sum_{j=1}^{m} E(\phi(O_j)).
\] (6.1)

Whenever defined, let \( \phi^{\text{BAL}}_{\text{OPT}} \) denote its optimal cost.

For example, in the trivial case of the identity cost function \( \phi(x) = x \) and no overflow list (\( \gamma = 0 \)), \( \phi^{\text{BAL}}_{\text{OPT}} \) corresponds to the average load per bucket, which is exactly \( r \), no matter what insertion algorithm or hash functions are used.

### 6.5 Theoretical Lower Bounds

We next show a lower bound on the achievable value of the optimal cost \( \phi^{\text{BAL}}_{\text{OPT}} \), as a function of the number of buckets \( m \), the number of elements \( n \), the average number of bucket accesses \( a \), and the overflow fraction \( \gamma \). (The proof appears in 6.5).

The lower bound is derived using a modified offline setting. In this setting, each bucket access is considered as a distinct element, as if initially \( a \cdot n \) distinct elements were hashed to the buckets, using a single hash function each. After storing all elements, we conceptually choose exactly \( (a - 1 + \gamma) \cdot n \) of the elements in a way that minimizes the cost function \( \phi^{\text{BAL}} \), resulting in exactly \( (1 - \gamma) \cdot n \) elements in the buckets. Since the cost function \( \phi \) is convex, then the marginal cost is the largest in the most occupied buckets. Therefore, a cost-minimizing removal process would remove element by element, picking the next element to remove in the most occupied bucket at each time.

Since we picked these elements in an offline manner, we necessarily perform better than any online setting. Thus, we bound the achievable value of \( \phi^{\text{BAL}}_{\text{OPT}} \).

**Theorem 18** When all hash functions are uniform, the optimal expected limit balancing cost \( \phi^{\text{BAL}}_{\text{OPT}} \) in the optimal access-constrained balancing problem is lower-bounded by

\[
\phi^{\text{BAL}}_{\text{LB}} = \sum_{j=0}^{k_0+1} P_{\text{LB}}(i) \cdot \phi(i),
\] (6.2)

where \( k_0 \) is the largest integer such that

\[
\frac{a \cdot r \cdot \Gamma(k_0, a \cdot r)}{(k_0 - 1)!} + k_0 \cdot \left(1 - \frac{\Gamma(k_0 + 1, a \cdot r)}{k_0!}\right) < r(1 - \gamma),
\] (6.3)

\( \Gamma(s, x) = \int_x^\infty t^{s-1} e^{-t} dt \) is the upper incomplete gamma function, and \( P_{\text{LB}} \) is a specific distribution that depends only on \( a, r, \) and \( \gamma \), but does not depend on the cost function \( \phi \).
Specifically, the distribution $P_{lb}$ is defined as follows:

$$P_{lb}(i) = \begin{cases} 
  e^{-a \cdot r(i)} & 0 \leq i < k_0 \\
  e^{-a \cdot r(k_0)} + c_0 + k_0 + 1 - k_0 p_0 - p_0 - r \cdot (1 - \gamma) & i = k_0 \\
  c_0 - k_0 + k_0 p_0 + r \cdot (1 - \gamma) & i = k_0 + 1 \\
  0 & \text{otherwise}
\end{cases} \quad (6.4)$$

where $c_0 = \frac{a \cdot r \cdot \Gamma(k_0, a \cdot r)}{(k_0 - 1)!}$, and $p_0 = \frac{\Gamma(k_0 + 1, a \cdot r)}{(k_0)!}$.

**Proof.** We derive the lower bound on the balancing by computing the best-case distribution of each bucket in an offline setting. We assume that whenever an hash function points to some bucket, an element is inserted into this bucket, having a total of $a \cdot n$ elements at the end of the process. Then, we remove exactly $(a - 1 + \gamma) \cdot n$ elements, which results in $(1 - \gamma) \cdot n$ total elements in the buckets. We remove the elements in a way that minimizes the cost function $\phi^{BAL}$.

In fact, by the convexity of the cost function $\phi$, minimizing the total cost $\phi^{BAL}$ can be done by removing the $(a - 1 + \gamma) \cdot n$ elements greedily, each time from one of the most occupied buckets. This is because the marginal cost is the largest (due to convexity) in those buckets. In the sequel, we relate to this process as the removal process.

We consider every hash value as a distinct element. Therefore, the number of elements (out of total $a \cdot n$ elements) that are mapped to bucket $j \in B$ follows a Binomial distribution with $a \cdot n$ independent experiments and a success probability of $\frac{1}{m}$. Let $Q_j(i) = \left(\frac{a \cdot n}{i}\right) \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{a \cdot n - i}$ denote the probability that bucket $j$ stores $i$ elements before the removal process.

Let $M_j(i)$ be the probability that bucket $j$ stores $i$ elements after the removal process. As we show now, $M_j(i)$ has to satisfy two constraints. First, since in the removal process elements are only removed (and not inserted), then the probability that some bucket stores less than $i$ elements after the removal process cannot be larger than before the removal process. Thus, for every $i$:

$$\sum_{k=0}^{i} M_j(k) \geq \sum_{k=0}^{i} Q_j(k). \quad (6.5)$$

Second, since we end up with exactly $(1 - \gamma) \cdot n$ elements, then:

$$\sum_{k=1}^{m} \left(\sum_{i=0}^{\infty} i \cdot M_k(i)\right) = (1 - \gamma) \cdot n, \quad (6.6)$$

that is, the expected number of elements in all the buckets after the removal process must be $(1 - \gamma) \cdot n.$

81
As we are looking for a lower bound on the balancing cost, our goal is to pick the bucket distribution that minimizes that cost. Consider bucket $j$ and assume that the expected occupancy after the removal process is $E_0$. Since all hash functions are uniform, by symmetry $E_0$ must be at most $\frac{an}{m}$. We construct the following distribution that minimizes the balancing cost of bucket $j$. The idea is to keep the original probabilities for low values of buffer occupancies, until the point where the expected occupancy $E_0$ is reached. On this point, we share the remaining probabilities such that we get the exact expected occupancy. Specifically, let $k_0$ be the largest integer such that

$$\sum_{i=0}^{k_0} i \cdot Q_j (i) + k_0 \cdot \left( 1 - \sum_{i=0}^{k_0} Q_j (i) \right) < E_0.$$  

That is, $k_0$ is the buffer occupancy until which we keep the original probability. Let $e_0 = \sum_{i=0}^{k_0} i \cdot Q_j (i)$ and $p_0 = \sum_{i=0}^{k_0} Q_j (i)$. In the sequel, we use $e_0$ and $p_0$ to construct the remainder of the distribution, that is, the probability for buffer occupancies $k_0$ and $k_0 + 1$.

We define the following distribution $P_j (i)$:

$$P_j (i) = \begin{cases} 
Q_j (i) & 0 \leq i < k_0 \\
Q_j (i) + e_0 + k_0 + 1 & i = k_0 \\
-k_0p_0 - p_0 - E_0 & i = k_0 + 1 \\
-e_0 - k_0 + k_0p_0 + E_0 & i = k_0 + 1 \\
0 & \text{otherwise}
\end{cases}$$

$P_j (i)$ satisfies both constraints from Equations (6.5) and (6.6). First, since we kept the original probabilities until buffer occupancy $k_0$, and then shared the remaining probabilities between $k_0$ and $k_0 + 1$, then for every $i$, $\sum_{k=0}^{k_0} P_j (k) \geq \sum_{k=0}^{k_0} Q_j (k)$. Second, let $\tilde{P}_j (i)$ be the random variable that corresponds to the distribution $P_j (i)$. Then, the expected number of elements in bucket $j$ is:

$$\mathbb{E} (\tilde{P}_j (i)) = \sum_{i=0}^{k_0+1} i \cdot P_j (i)$$

$$= e_0 - k_0Q_j (k_0) + k_0P_j (k_0) + (k_0 + 1) P_j (k_0 + 1)$$

$$= E_0$$

Thus, $P_j (i)$ satisfies the two constraints.

We now show that it minimizes the cost function, over all distributions that satisfy both constraints. Let $G_j (i)$ be a distribution over the buffer occupancies after the removal process that satisfies both constraints. Let $i_0$ be the smallest integer such that $G_j (i_0) \neq P_j (i_0)$; if such $i_0$ does not exist, we are done since $G_j (i)$ coincides with $P_j (i)$. We will show that $G_j (i_0) > P_j (i_0)$. Also, let $i_1$ be the largest integer such that $G_j (i_1) > P_j (i_1)$.

82
We now show that if \( i_0 \) and \( i_1 \) are defined, then \( G_j(i_0) > P_j(i_0) \), and \( i_1 - i_0 \geq 2 \). We distinguish between 3 cases: \( i_0 > k_0, i_0 = k_0 \) and \( i_0 < k_0 \).

First, in case of \( i_0 > k_0 \), for every bucket occupancy \( i \leq k_0, G_j(i) = P_j(i) \). Thus, \( G_j(i) = P_j(i) \) for every \( i \), as \( G_j(i) \) satisfies the second constraint (Equation (6.6)), implying that \( i_0 \) and \( i_1 \) are not defined.

In case \( i_0 < k_0 \), by the first constraint (Equation (6.5)) and the fact that \( P_j(i) = Q_j(i) \) for every \( i < i_0 \), we get that \( G_j(i_0) > P_j(i_0) \). We now show that \( i_1 > k_0 \), implying that \( i_1 - i_0 \geq 2 \). Assume on the contrary that \( i_1 \leq k_0 \), then \( G_j(k_0 + 1) \leq P_j(k_0 + 1) \) and for every \( i > k_0 + 1, G_j(i) = 0 \). Let \( \tilde{G}_j(i) \) be the random variable that corresponds to \( G_j(i) \). Since \( \mathbb{E}\{\tilde{G}_j(i)\} = \mathbb{E}\{\tilde{P}_j(i)\} \) and for any random variable \( X \) that takes values in \( \mathbb{N} \), \( \mathbb{E}(X) = \sum_{\ell=1}^{\infty} \Pr\{X \geq \ell\} \), we get that

\[
\sum_{i=1}^{k_0+1} \sum_{\ell=i}^{\infty} G_j(\ell) = \sum_{i=1}^{k_0+1} \sum_{\ell=i}^{\infty} P_j(\ell). \tag{6.10}
\]

Thus,

\[
\sum_{i=1}^{k_0+1} \left[ \sum_{\ell=i}^{\infty} G_j(\ell) - \sum_{\ell=i}^{\infty} P_j(\ell) \right] = 0. \tag{6.11}
\]

By the definition of \( P_j(i) \), for every \( i \leq k_0 \), \( \sum_{\ell=i}^{\infty} P_j(\ell) = \sum_{\ell=i}^{\infty} Q_j(\ell) \). So,

\[
\sum_{i=1}^{k_0} \left[ \sum_{\ell=i}^{\infty} G_j(\ell) - \sum_{\ell=i}^{\infty} Q_j(\ell) \right] + G_j(k_0 + 1) - P_j(k_0 + 1) = 0. \tag{6.12}
\]

The first constraint (Equation (6.5)) states that \( \sum_{\ell=0}^{i} G_j(\ell) \geq \sum_{\ell=0}^{i} Q_j(\ell) \), thus, \( \sum_{\ell=i}^{\infty} G_j(\ell) \leq \sum_{\ell=i}^{\infty} Q_j(\ell) \). Also, we know that \( G_j(k_0 + 1) \leq P_j(k_0 + 1) \). Since \( G_j(i_0) \neq Q_j(i_0) \), we get that the total sum cannot be zero, that is, at least one element in the sum is negative (but none is positive). Therefore, \( i_1 > k_0 \).

The last case to consider is when \( i_0 = k_0 \). If \( G_j(i_0) < P_j(i_0) \), then \( G_j(i) \) clearly does not satisfy the second constraint (Equation (6.6)) as for every \( i < k_0, G_j(i_0) = P_j(i_0) \). Therefore, \( G_j(i_0) > P_j(i_0) \). Furthermore, the second constraint implies that there must be some integer \( i_1 > k_0 + 1 \) such that \( G_j(i_1) \neq 0 \). Therefore, \( i_1 - i_0 \geq 2 \).

We are now ready to define another distribution, \( G'_j(i) \), which also has minimal cost function:

\[
G'_j(i) = \begin{cases} 
G_j(i) - w & i \in \{i_0, i_1\} \\
G_j(i) + w & i \in \{i_0 + 1, i_1 - 1\} \\
G_j(i) & \text{otherwise}
\end{cases} \tag{6.13}
\]

where \( w = \min\{G_j(i_0) - P_j(i_0), G_j(i_1) - P_j(i_1)\} \). Notice that \( G'_j(i) \) is well-defined since \( i_1 - i_0 \geq 2 \). In addition, \( w > 0 \) since \( G_j(i_0) > P_j(i_0) \) and \( G_j(i_1) > P_j(i_1) \). Hence, \( G'_j(i) \), which clearly preserves both constraints, has a cost no larger than \( G_j(i) \). By continuing
this process, we end up with \( P_j(i) \) no matter what \( G_j(i) \) is, as \( i_1 - i_0 \) decreases at each step by at least 1. This implies that \( P_j(i) \) minimizes the cost function.

Finally, since we are interested in the limit balancing cost lower bound \( \phi_{\text{BAL}} \), we consider the limit distribution \( P_{\text{lb}}(i) \) of the distribution \( P_j(i) \) that was found to be optimal for any finite parameters. This is done by using the Poisson approximation for the binomial distribution \( Q_j(i) \) of the buckets occupancy before the removal process [8, 91, 92], where we use the same approximation to find the values of \( k_0, p_0 \) and \( e_0 \). Also, by symmetry we get that \( E_0 = \frac{(1-\gamma)n}{m} = r \cdot (1 - \gamma) \). \( \square \)

Interestingly, the element-elimination algorithm does not depend on the precise convex cost function \( \phi \). This is why the resulting bucket-load distribution \( P_{\text{lb}} \) is independent of \( \phi \) as well.

In addition, this distribution \( P_{\text{lb}} \) is defined on a compact space. As a result, it can also be shown that if a sequence of cost functions \( \{ \phi_k \} \) converges pointwise to some cost function \( \phi \), then the sequence of lower bounds converges as well to the corresponding lower bound on \( \phi \). This can then be used to extend the cost functions to the maximum-load metric commonly used in the literature [24, 25, 26].

Fig. 6.2 shows the lower-bound distribution \( P_{\text{lb}}(i) \) for load \( r = \frac{n}{m} = 8 \), overflow fraction \( \gamma = 0 \) and average access rate \( a \in \{1, 1.1, 1.2\} \). Note that when \( a = 1 \), all elements use a single lookup, and therefore there is no element elimination in the offline algorithm. The distribution \( P_{\text{lb}} \) simply follows a Poisson distribution with parameter \( \lambda = r \), as shown using the solid line. Then, for larger values of \( a \), the element elimination algorithm reduces the probability of having a large bin load.

We also consider a setting where \( \ell \leq d \) different distributions over the buckets are used by the \( d \) hash functions. Denote these distributions by \( f^1, \ldots, f^\ell \), and assume that distribution \( f^i \) is used by a fraction \( k_i \) of the total bucket accesses, with \( \sum_{i=1}^{\ell} k_i = 1 \). We now show that Theorem 18 holds also in this case when \( \sum_{p=1}^\ell k_p f_p(i) = \frac{1}{m} \).
Theorem 19 If $\sum_{p=1}^{\ell} k_p f_p(i) = \frac{1}{m}$ then the optimal expected limit balancing cost $\phi_{\text{bal}}^{\text{opt}}$ in the optimal access-constrained balancing problem has the same lower bound as in Theorem 18.

Proof. The number of elements mapped by the hash functions with distribution $f_p$ to bucket $i$ follows approximately a Poisson distribution with rate $k_p \cdot a \cdot n \cdot f_p(i)$ (see proof of Theorem 18). Since the sum of Poisson random variables is also a Poisson random variable, the total number of elements mapped to the bucket $i$ follows a Poisson distribution with rate $an \sum_{p=1}^{\ell} k_p f_p(i)$.

Thus, the proof of Theorem 18 implies that if for every bucket $i$, this rate equals $an$, we get the same limit lower bound balancing cost $\phi_{\text{lb}}^{\text{bal}}$.

$\square$

6.6 SINGLE-B - A Single-Choice Balancing Scheme

We have found a lower-bound for the optimal cost. In the sequel, we focus on finding values of $a$ and $\gamma$ in which we can match this bound.

For better intuition, we start by analyzing a simplistic balancing scheme, denoted SINGLE-B, that is associated with 2 parameters $\bar{h}$ and $p$. This scheme only uses a single uniformly-distributed hash function $H$. Each element is stored in bucket $H(x)$ if it has less than $\bar{h}$ elements. In case there are exactly $\bar{h}$ elements, the element is stored in the bucket with probability $p$ and in the overflow list with probability $1-p$. Otherwise, the element is stored in the overflow list.

6.6.1 Description by Differential Equations

As in Chapter 4, we use a deterministic system of differential equations [47, 23, 8, 5]. In this section, we provide a succinct description of SINGLE-B.

Recall that in this approach, we consider the element insertion process as performed between the time $t = 0$ and $t = 1$, that is, at time $t = \frac{j}{n}$ the $j$-th element is inserted. Furthermore, let $F_i(\frac{j}{n})$ denote the fraction of buckets in the hash table that store exactly $i$ elements at time $\frac{j}{n}$, just before element $j$ is inserted, and $\vec{F}(\frac{j}{n})$ be the vector of all $F_i(\frac{j}{n})$’s. Also, let $\Delta F_i(\frac{j+1}{n}) \triangleq F_i(\frac{j+1}{n}) - F_i(\frac{j}{n})$ denote the change in the fraction of buckets that store exactly $i$ elements between times $\frac{j}{n}$ and $\frac{j+1}{n}$. Then

$$
\mathbb{E} \left( \Delta F_i(\frac{j+1}{n}) \mid F_i(\frac{j}{n}) \right) = \begin{cases} 
-\frac{1}{m} F_0(\frac{j}{n}) & i = 0 \\
\frac{1}{m} \left( F_{\bar{h}-1}(\frac{j}{n}) - p \cdot F_{\bar{h}}(\frac{j}{n}) \right) & i = \bar{h} \\
\frac{1}{m} \cdot p \cdot F_{\bar{h}-1}(\frac{j}{n}) & i = \bar{h} + 1 \\
\frac{1}{m} \left( F_{i-1}(\frac{j}{n}) - F_i(\frac{j}{n}) \right) & \text{otherwise}
\end{cases}
$$ (6.14)
At time \( t = 0 \), \( F_i(0) = 1 \) if \( i = 0 \) and 0 otherwise.

The probability that element \( j \) hits a bucket storing \( i \) elements is \( F_i \left( \frac{j}{n} \right) \). Thus, in the first equation, the fraction of empty buckets decreases when element \( j \) reaches an empty bucket, which occurs with probability of \( F_0 \left( \frac{j}{n} \right) \). Likewise, in the second equality, the fraction of buckets that store \( \bar{h} \) elements increases when element \( j \) hits a bucket storing \( \bar{h} - 1 \) elements (with probability of \( F_{\bar{h}-1} \left( \frac{j}{n} \right) \)), and decreases with probability of \( p \) when the element hits a bucket storing \( \bar{h} \) elements (with total probability of \( p \cdot F_{\bar{h}} \left( \frac{j}{n} \right) \)). In the third equality, the fraction of buckets storing \( \bar{h} + 1 \) elements increases if element \( j \) hits a bucket storing \( \bar{h} \) elements. Last, in all other cases, the fraction of buckets storing \( i \) elements increases if element \( j \) hits a bucket storing \( i - 1 \) elements, and decreases if it hits a bucket storing \( i \) elements. Any such increment or decrement is by a value of \( \frac{1}{m} \), thus, all equations are multiplied by \( \frac{1}{m} \).

By dividing both sides of the equation by \( \frac{1}{n} \) and considering the fact that \( n \) is large, so that the values of \( \Delta F_i \left( \frac{i+1}{n} \right) \) are comparatively very small, we can use the fluid limit approximation, which is often very accurate [47]:

$$\frac{df_i(t)}{dt} = \begin{cases} 
-\frac{n}{m} f_0(t) & i = 0 \\
\frac{n}{m} (f_{i-1}(t) - p \cdot f_i(t)) & i = \bar{h} \\
p \cdot \frac{n}{m} f_{\bar{h}-1}(t) & i = \bar{h} + 1 \\
\frac{n}{m} (f_{i-1}(t) - f_i(t)) & \text{otherwise}
\end{cases}$$

(6.15)

More formally, let \( \tilde{f}(t) \triangleq (f_1(t), \ldots, f_d(t)) \) be the solution of the above set of linear differential equations when assuming \( f_0(0) = 1 \) and \( f_i(0) = 0 \) for each \( i \neq 0 \). Then, by Kurtz theorems [93, 94, 95], the probability that \( \tilde{f} \) deviates from \( \tilde{F} \) by more than some constant \( \varepsilon \) decays exponentially as a function of \( n \) and \( \varepsilon^2 \) [47]. For further intuition behind this statement, refer to [47] and [96, Chapter 3.4].

Fig. 6.3 shows the evolution over time of \( f_0, \ldots, f_3 \) where \( r = 2.5 \), \( p = 0.5 \) and \( \bar{h} = 2 \), comparing the model with simulated values. In the simulation we used \( n = 25,000 \), and so \( m = 10,000 \).

6.6.2 Optimality Result

In this section, we solve the system of differential equations yielding the following optimality result.

**Theorem 20** Consider the single-b balancing scheme with \( m \) buckets and \( n \) elements, and use the notations of \( k_0 \), \( p_0 \), \( e_0 \) and \( P \) from Theorem 18. Then for any value of \( \gamma \), the single-b scheme solves the optimal access-constrained balancing problem for \( a = 1 \) whenever it satisfies the two following conditions:

(i) \( \bar{h} = k_0 \);

(ii) \( \bar{h} = k_0 \).

86
(ii) \emph{p is the solution of the following fixed-point equation:}
\[
\frac{e^{-pt}}{(1-p)^h} - \frac{e^{-r}}{(1-p)^h} \sum_{i=0}^{h-1} \frac{(r(1-p))^i}{i!} = P(k_0).
\]

\textbf{Proof.} We solve the differential equations one by one, substituting the result of the equation for \(\frac{df(i)}{dt}\) into the equation for \(\frac{df_{i+1}(t)}{dt}\). The first equation depends only on \(f_0(t)\), and we get immediately that \(f_0(t) = e^{-\frac{t}{pt}}\), or \(f_0 = e^{-r\cdot t}\). Each equation for \(\frac{df_i(t)}{dt}\), where \(i \leq h\), depends only on \(f_{i-1}(t)\) and \(f_i(t)\), and we get that for \(i < h\), \(f_i(t) = \frac{1}{i!}(r \cdot t)^i e^{-r\cdot t}\).

For \(f_h(t)\), we get that for \(0 \leq p < 1\)
\[
f_h(t) = \frac{e^{-pt}}{(1-p)^h} - \frac{e^{-r}}{(1-p)^h} \sum_{i=0}^{h-1} \frac{(r \cdot t \cdot (1-p))^i}{i!}
\]  
(6.16)
and for \(p = 1\),
\[
f_h(t) = \frac{1}{h!}(r \cdot t)^h e^{-r\cdot t}.
\]  
(6.17)

We also use the fact that \(\sum_{i=0}^{h+1} f_i = 1\) to get \(f_{h+1}(t)\).

By substituting \(t = 1\) in \(f_i(t)\), for \(i < h\), we find that \(f_i(1) = \frac{1}{i!}(r)^i e^{-r}\). We note that it is also the probability that an arbitrary bucket stores \(i\) elements, and that it is equal to \(P_{lb}(i)\), thus mimicking the distribution of \(P_{lb}(i)\) for \(i < h\).

We are left to show that there exists such a \(p \in [0, 1]\) so that using its value for \(f_h(1)\) will result in the exact expression for \(P_{lb}(h)\). When substituting \(p = 0\), we get that \(f_h(1) = 1 - e^{-r} \sum_{i=0}^{h-1} \frac{r^i}{i!}\) which is clearly larger than \(P_{lb}(h)\) (it is equal when \(P_{lb}(h+1) = 0\)). On the other hand, when substituting \(p = 1\), we get that \(f_h(1) = \frac{1}{h!}(r)^h e^{-r}\) which is lower than \(P_{lb}(h)\). Thus, using the Intermediate Value Theorem (all functions are clearly continuous), there exists some \(p \in [0, 1]\) such that \(f_h(1) = P_{lb}(h)\). Since \(\sum_{i=0}^{h+1} f_i(1) = \sum_{i=0}^{h+1} P_{lb}(i) = 1\), we get also that \(f_{h+1}(1) = P_{lb}(h+1)\). \(\square\)
6.7 SEQUENTIAL-B - A Multiple-Choice Balancing Scheme

We now introduce the SEQUENTIAL-B scheme, which is also associated with two parameters \( \bar{h} \) and \( p \). In the SEQUENTIAL-B scheme, we use an ordered collection of \( d \) hash functions \( \mathcal{H} = \{ H_1, \ldots, H_d \} \), such that all functions are independent and uniformly distributed. Upon inserting an element \( x \), the scheme successively reads the buckets \( H_1(x), H_2(x), \ldots, H_d(x) \), and places \( x \) in the first bucket that satisfies one of the following two conditions: (i) the bucket stores less than \( \bar{h} \) elements, or, (ii) the bucket stores exactly \( \bar{h} \) elements, and \( x \) is inserted with probability \( p \). If the insertion algorithm fails to store the element in all the \( d \) buckets, \( x \) is stored in the overflow list. Last, to keep an average number of bucket accesses per element of at most \( a \), the process stops when a total of \( a \cdot n \) bucket accesses has been reached; the remaining elements are placed in the overflow list.

Using the approach presented in Section 6.6.1, we now describe the dynamics of the SEQUENTIAL-B scheme as a system of differential equations.

6.7.1 Description by Differential Equations

We start analyzing the SEQUENTIAL-B scheme by first assuming that there is no constraint on the total number of bucket accesses (that is, \( a = \infty \)) and characterizing the dynamics of the scheme as a system of differential equations.

As before, let \( f_i(t) \) represent the fraction of buckets storing \( i \) elements at time \( t \), then

\[
\frac{df_i(t)}{dt} = \begin{cases} 
- \frac{n}{m} \cdot f_0 \cdot g(t) & i = 0 \\
\frac{n}{m} \cdot (f_{i-1}(t) - p \cdot f_i(t)) \cdot g(t) & i = \bar{h} \\
\frac{n}{m} \cdot p \cdot f_{\bar{h}}(t) \cdot g(t) & i = \bar{h} + 1 \\
\frac{n}{m} \cdot (f_{i-1}(t) - f_i(t)) \cdot g(t) & \text{otherwise}
\end{cases}
\]

for \( i = 0, 1, \ldots, \bar{h} \), with

\[
g(t) = \sum_{k=0}^{d-1} \left( (1-p) \cdot f_{\bar{h}}(t) + f_{\bar{h}+1}(t) \right)^k \\
= \frac{1 - ((1-p) \cdot f_{\bar{h}}(t) + f_{\bar{h}+1}(t))^d}{1 - ((1-p) \cdot f_{\bar{h}}(t) + f_{\bar{h}+1}(t))},
\]

where \( f_0(0) = 1 \) and \( f_i(0) = 0 \) for each \( i \neq 0 \) as an initial condition. Comparing with the differential equations of the SINGLE-B scheme (Equation (6.14)), there is an additional factor of \( g(t) \). For instance, in the first equation, \( f_0(t) \) is replaced by \( f_0(t) \cdot g(t) = \sum_{k=0}^{d-1} \left[ (1-p) \cdot f_{\bar{h}}(t) + f_{\bar{h}+1}(t) \right]^k \cdot f_0(t) \), which represents the sum of the probabilities of mapping to an empty bucket after being mapped \( k = 0, 1, \ldots, d-1 \) times to a bucket in which the element could not be stored; in other blocks, each bucket either already had \( \bar{h}+1 \) elements, or had \( \bar{h} \) element but with probability of \( 1 - p \) the element was
denied insertion.

We are also interested in the average number of bucket accesses performed during this process. Let \( f_a^{\text{sequential-b}}(t) \) denote the cumulative number of bucket accesses performed by time \( t \), normalized by \( n \). It can be modeled as

\[
\frac{df_a^{\text{sequential-b}}(t)}{dt} = \sum_{k=1}^{d-1} k \cdot (f_n(t))^{k-1}(1 - f_n(t)) + d \cdot (f_n(t))^{d-1},
\]

(6.20)

where

\[
f_n(t) = \left( (1 - p) \cdot f_h(t) + f_{h+1}(t) \right);
\]

and \( f_a^{\text{sequential-b}}(0) = 0 \) as an initial condition.

The differential equation reflects the fact that at a given time \( t \), the cumulative number of bucket accesses increases by \( 1 \leq k < d \) bucket accesses whenever in the first \( k - 1 \) bucket accesses an element is not stored and in the next one the element is stored. It also increases by \( d \) bucket accesses whenever in the first \( d - 1 \) bucket accesses an element is not stored, independently of the bucket state in the \( d \)-th bucket access.

6.7.2 Optimality Result

The resulting system of equations is hard to solve analytically, implying it cannot be used directly to show optimality (naturally, the system can be solved numerically to provide a numerical approximation of the expected balancing cost).

Therefore, we take a different approach. We analyze the \textit{sequential-b} scheme by reducing it to the \textit{single-b} scheme: Since both the \textit{single-b} and \textit{sequential-b} schemes use the same uniform distribution, a new attempt to insert an element after an unsuccessful previous attempt in the \textit{sequential-b} scheme is equivalent to creating a new element in the \textit{single-b} scheme and then trying to insert it. In other words, the number of elements successfully inserted by the \textit{sequential-b} scheme after considering \( n \) elements and using a total of \( a \cdot n \) bucket accesses is the same as the number of elements successfully inserted by the \textit{single-b} scheme after considering \( a \cdot n \) elements.

\textbf{Theorem 21} Consider the \textit{sequential-b} balancing scheme with \( m \) buckets and \( n \) elements, and use the notations of \( k_0, p_0, e_0 \) and \( P \) from Theorem 18. The \textit{sequential-b} scheme solves the optimal access-constrained balancing problem whenever it satisfies the three following conditions:

(i) \( \bar{h} = k_0 \);

(ii) all \( a \cdot n \) memory accesses are exhausted before or immediately after trying to insert the \( n \)-th element;

(iii) \( p \) is the solution of the following fixed-point equation:

\[
\frac{e^{-\alpha r}}{(1-p)^{r}} - \frac{e^{-\alpha r}}{(1-p)^{k}} \sum_{i=0}^{k-1} \frac{(\alpha r - (1-p))^{i}}{i!} = P(k_0).
\]

Moreover, the optimality region is given by the overflow list of size \( \gamma_0 \cdot n \) that results in
exhausting all $a \cdot n$ memory immediately after trying to insert the $n$-th element.

Proof. We compare the sequential-B scheme with the single-B scheme. In the sequential-B scheme we continually try to insert each element, until either it is placed or all $d$ functions are used.

Note that all hash functions have the same (uniform) distribution over all buckets. Thus, for every $i$, $f_i(t)$ are independent of the exact elements that are hashed. Therefore, applying $d_1 \leq d$ hash functions on the same element is equivalent to applying a single hash function on $d_1$ elements. This implies we can use the results of the single-B scheme, in which a total of $a \cdot n$ elements are considered and therefore a total of $a \cdot n$ bucket accesses are performed.

Given an average number of bucket accesses $a$, we set $\bar{h} = k_0$, and let sequential-B operate until $a$ is reached, that is, until time $t_0 \leq 1$ such that $f_{\text{sequential-B}}^a(t_0) = a$. We apply the single-B dynamics to get $f_i(t_0)$, and get that

$$f_i(t_0) = \begin{cases} \frac{1}{\pi} (a \cdot r)^i e^{-a \cdot r} & \text{if } i < \bar{h} \\ \frac{e^{-a \cdot r}}{(1-p)^r} - \frac{e^{-a \cdot r}}{(1-p)^r} \sum_{j=0}^{\bar{h}-1} \frac{(a \cdot r - 1)(1-p)^r}{j!} & \text{if } i = \bar{h} \\ 1 - \sum_{j=0}^{\bar{h}-1} \frac{1}{\pi} (a \cdot r)^j e^{-a \cdot r} - \frac{e^{-a \cdot r}}{(1-p)^r} + \frac{e^{-a \cdot r}}{(1-p)^r} \sum_{j=0}^{\bar{h}-1} \frac{(a \cdot r - 1)(1-p)^r}{j!} & \text{if } i = \bar{h} + 1 \end{cases}$$

Using the same consideration as in the proof of Theorem 20, we find that there is some $p \in [0, 1]$ such that $f_i(t_0)$ is exactly $P_{\text{LB}}(i)$, for every $i$.

However, note that the sequential-B scheme cannot bring to any desired number of bucket accesses $a$, and is limited to $f_{\text{sequential-B}}^a(1)$. Since $f_{\text{sequential-B}}^a(t)$ increases as $k_0$ decreases, and $k_0$ decreases as $\gamma$ increases, then the minimum $\gamma$ such that the sequential-B scheme achieves optimality, for a given $a$ and $r$, is given by $\gamma_0$ such that $f_{\text{sequential-B}}^a(1) = a$.

Thus, we get the optimality region of this scheme.

\[ \square \]

6.8 The Multi-Level Hash Table (MHT-B) Balancing Scheme

The multi-level hash table (MHT-B) balancing scheme conceptually consists of $d$ separate subtables $T_1, \ldots, T_d$, where $T_i$ has $a_i \cdot n$ buckets, and $d$ associated hash functions $H_1, \ldots, H_d$, defined such that $H_i$ never returns values of bucket indices outside $T_i$.

This scheme is similar to the mht scheme presented in previous chapter. However, it has a slight variation as described below.

Using the MHT-B scheme, element $x$ is placed in the smallest $i$ that satisfies one of the following two conditions: (i) the bucket $H_i(x)$ stores less than $\bar{h}$ elements, or, (ii) the bucket $H_i(x)$ stores exactly $\bar{h}$ elements, and element $x$ is then inserted with probability $p$.

If the insertion algorithm fails to store the element in all the $d$ tables, $x$ is placed in the overflow list. Since that smallest $i$ with available space is used, the bucket accesses for each element $x$ are sequential, starting from $H_1(x)$ until a place is found or all $d$ hash functions are used (and the element is stored in the overflow list).
We now give the description of the dynamics of the MHT-B scheme using a system of differential equations.

### 6.8.1 Description by Differential Equations

The system of differential equations that characterizes the dynamics of MHT-B is influenced by the static partitioning of the memory among subtables, which introduces extra variables. Specifically, let \( f_{i,j}(t) \) be the fraction of buckets in subtable \( T_j \) that store exactly \( i \) elements. Then:

\[
\frac{df_{i,j}(t)}{dt} = \begin{cases} 
-\frac{n}{a_j m} f_{0,j}(t) g_j(t) & i = 0 \\
\frac{n}{a_j m} (f_{i-1,j}(t) - p \cdot f_{i,j}(t)) g_j(t) & i = \bar{h} \\
\frac{n}{a_j m} (f_{i+1,j}(t) - f_{i,j}(t)) g_j(t) & i = \bar{h} + 1 \\
\frac{n}{a_j m} (f_{i-1,j}(t) - f_{i,j}(t)) g_j(t) & \text{otherwise}
\end{cases}
\]

(6.23)

where

\[
g_j(t) = \prod_{k=1}^{j-1} \left( (1-p) f_{\bar{h},k}(t) + f_{\bar{h}+1,k}(t) \right)
\]

(6.24)

represents the probability that all the insertion attempts in subtables \( T_1, \ldots, T_{j-1} \) do not result in storing the element, and thus that MHT-B will attempt to insert the element in subtable \( T_j \). By convention \( g_1(t) = 1 \). The initial conditions are \( f_{i,j}(0) = 1 \) for \( i = 0 \) and \( f_{i,j}(0) = 0 \) otherwise.

As in the SEQUENTIAL-B scheme, let \( f^a_{\text{MHT-B}}(t) \) denote the cumulative number of bucket accesses done by time \( t \), normalized by \( n \). Then the following differential equation reflects the dynamics of \( f^a_{\text{MHT-B}}(t) \):

\[
\frac{df^a_{\text{MHT-B}}(t)}{dt} = d \cdot g_d(t) + \sum_{k=1}^{d-1} k \cdot g_k(t) \left( 1 - (1-p) f_{\bar{h},k}(t) - f_{\bar{h}+1,k}(t) \right),
\]

(6.25)

with \( f^a_{\text{MHT-B}}(0) = 0 \).

This description of the dynamics of the MHT-B scheme using a system of differential equations is difficult to solve. Our approach relies on the fact that each subtable follows a local SINGLE-B scheme. More specifically, all elements attempting to access some subtable \( T_j \) only access a single uniformly-distributed bucket in \( T_j \), and if this bucket is full, do not consider any other bucket in \( T_j \). Thus, within each subtable \( T_j \), MHT-B behaves like SINGLE-B, with a number of initial elements that depends on previous subtables.

Let \( f^a_{\text{SINGLE-B}}(t) \) be the fraction of buckets that store exactly \( i \) elements at time \( t \) in the
SINGLE-B scheme. As in the proof of Theorem 20, it is given by:

\[
f_i^{\text{SINGLE-B}} (t) = \begin{cases} 
\frac{1}{\pi} (r \cdot t)^i e^{-r \cdot t} & i < \bar{h} \\
\frac{e^{-r \cdot t}}{(1-p)^n} - \frac{e^{-r \cdot t}}{(1-p)^n} \sum_{j=0}^{\bar{h}-1} \frac{(r \cdot t - j)}{j!} & i = \bar{h} \\
1 - \sum_{j=0}^{\bar{h}-1} \frac{1}{\pi} (r \cdot t)^i e^{-r \cdot t} - \frac{e^{-r \cdot t}}{(1-p)^n} \sum_{j=0}^{\bar{h}-1} \frac{(r \cdot t - j)}{j!} & i = \bar{h} + 1
\end{cases}
\] (6.26)

Also, let \( \gamma^t_{\text{SINGLE-B}}(t) \) be the fraction of the elements that are not stored in the buckets out of all the elements that arrived up to time \( t \). Given all \( f_i^{\text{SINGLE-B}}(t) \)'s, it is simply the complementary of the expectation of the number of elements in the buckets up to time \( t \) normalized by the total number of elements arrived up to this time:

\[
\gamma^t_{\text{SINGLE-B}} (t) = 1 - \frac{m}{nt} \cdot \sum_{i=0}^{\bar{h}+1} f_i^{\text{SINGLE-B}} (t).
\] (6.27)

Let \( n_j(t) \) denote the number of elements that are considered in subtable \( T_j \) up to time \( t \), and \( \gamma^t_j(t) \) denote the fraction of these elements that are not placed in subtable \( T_j \). We will express these using \( f_i^{\text{SINGLE-B}} \) and \( \gamma^t_{\text{SINGLE-B}} \), the corresponding functions in the SINGLE-B scheme.

Note that as shown in Equations (6.26) and (6.27), \( f_i^{\text{SINGLE-B}}(t) \) and \( \gamma^t_{\text{SINGLE-B}}(t) \) only depend on the time \( t \), the number of elements \( n \), the number of buckets \( m \), the bucket size \( \bar{h} \) and the probability \( p \); thus, we refer to them as \( f_i^{\text{SINGLE-B}}(t,m,n,\bar{h},p) \) and \( \gamma^t_{\text{SINGLE-B}}(t,m,n,\bar{h},p) \). We obtain the following theorem, which is valid for any arbitrary partition of the subtables.

**Theorem 22** Consider an \((a,d,r)\) MHT-B balancing scheme in which for each \( 1 \leq j \leq d \), subtable \( T_j \) has \( \alpha_j \cdot m \) buckets, with \( \sum \alpha_j = 1 \). Then, as long as \( f_{\text{MHT-B}}(t) \leq a \), the functions \( n_j(t) \), \( \gamma^t_j(t) \) and \( f_{i,j}(t) \) satisfy

\[
n_j(t) = n \cdot t \cdot \prod_{k=1}^{j-1} \gamma^t_k(t),
\] (6.28)

\[
\gamma^t_j(t) = \gamma^t_{\text{SINGLE-B}}(1, \alpha_j m, n_j(t), \bar{h}, p),
\] (6.29)

\[
f_{i,j}(t) = f_i^{\text{SINGLE-B}}(1, \alpha_j m, n_j(t), \bar{h}, p).
\] (6.30)

**Proof.** By the definition of the MHT-B scheme, it follows immediately that \( n_j(t) = \gamma^t_{j-1}(t) n_{j-1}(t) \); since \( n_1(t) = n \cdot t \) (all elements go through the first subtable), we get that

\[
n_j(t) = n \cdot t \cdot \prod_{k=1}^{j-1} \gamma^t_k(t).
\]

Equations (6.29) and (6.30) are immediately derived by setting the right parameters for each SINGLE-B scheme within each subtable \( T_j \); namely, its total number of buckets is \( \alpha_j \cdot m \) and the number of elements by time \( t \) is \( n_j(t) \). \( \square \)
6.9 Optimality Result

The system of differential equations of the MHT-B scheme is generally too difficult to solve. As in SEQUENTIAL-B, this can be circumvented by relying on our results from the SINGLE-B scheme. We now show the optimality of the MHT-B scheme.

**Theorem 23** Consider an \(\langle a, d, r \rangle\) MHT-B balancing scheme in which each subtable \(T_j\) has \(\alpha_j \cdot m\) buckets, with \(\sum \alpha_j = 1\), and use the notations of \(k_0, p_0, e_0\) and \(P\) from Theorem 18. Further, let \(p(a)\) denote the overflow fraction of the SINGLE-B scheme with \(a \cdot n\) elements. Then, the \(\langle a, d, r \rangle\) MHT-B scheme solves the optimal access-constrained balancing problem whenever it satisfies the following four conditions:

(i) \(\bar{h} = k_0\);
(ii) all \(a \cdot n\) memory accesses are exhausted before or immediately after trying to insert the \(n\)-th element;
(iii) \(p\) is the solution of the following fixed-point equation:

\[
\frac{e^{-p \cdot r}}{(1-p)^h} - \frac{e^{-a \cdot r}}{(1-p)^h} \sum_{i=0}^{h-1} \frac{(a \cdot (1-p))^i}{i!} = P(k_0);
\]

(iv) the subtable sizes \(\alpha_j \cdot m\) follow a geometric decrease of factor \(p(a)\):

\[
\alpha_j = \left(1 - p(a)\right) \frac{1}{1 - p(a)} p(a)^{j-1}.
\]

Moreover, the optimality region is given by the overflow list of size \(\gamma \cdot n\) that results in exhausting all \(a \cdot n\) memory accesses immediately after trying to insert the \(n\)-th element. Furthermore, if all four conditions are met then all buckets have an identical occupancy distribution.

**Proof.** Given the average number of bucket accesses \(a\), we set \(\bar{h} = k_0\). We would like to let MHT-B operate until \(a\) is reached, that is, until time \(t_0 \leq 1\) such that \(f_{\text{MHT-B}}^a(t_0) = a\). However, \(f_{\text{MHT-B}}^a(t)\) depends on the subtables sizes \(\alpha_j\)'s.

Up to time \(t_0\), in which we aim to exhaust all \(a \cdot n\) bucket accesses, we used exactly \(n_j(t_0)\) times the hash function \(H_j(x)\) in subtable \(T_j\). Since we aim at an optimal balancing cost, the necessary condition on the distributions of the hash functions, given in Theorem 19, immediately implies that

\[
\alpha_j = \frac{n_j(t_0)}{n \cdot a}.
\]

By substituting the expression for \(\alpha_j\) in (6.28), we get:

\[
\text{Theorem 23.}\]
It is important to notice that, quite surprisingly, \( \gamma^f (t_0) \) does not depend on \( j \).

We now obtain the time \( t_0 \) by observing that \( n_j (t_0) = n \cdot t_0 \cdot p (a)^{j-1} \), thus \( \alpha_j = \frac{t_0 \cdot p (a)^{j-1}}{a} \).
Since \( \sum_{k=1}^d \alpha_k = 1 \), we get \( \sum_{k=1}^d \frac{t_0 p^{j-1}}{a} = 1 \), and therefore \( t_0 \) is given by the sum of a geometric series:

\[
t_0 = a \left( \frac{1 - p (a)}{1 - p (a)^a} \right). \tag{6.33}
\]

This, in turn, immediately gives us the claimed memory partitioning \( \alpha_j \).

We now turn to show that all bucket distributions are identical. In subtable \( T_j \), the total number of elements considered is \( n_j (t_0) = n \cdot t_0 \cdot p (a)^{j-1} \), while there are \( m \cdot \alpha_j = m \cdot \left( \frac{1 - p (a)}{1 - p (a)^a} \right) p (a)^{j-1} \) buckets. Hence, by Theorem 22, we get that the fraction of buckets in subtable \( T_j \) that store exactly \( i \) elements \( f_{i,j} (t) \) is given by

\[
f_i^{\text{SINGLE-B}} \left( 1, m \cdot \left( \frac{1 - p (a)}{1 - p (a)^a} \right) p (a)^{j-1}, n \cdot t_0 \cdot p (a)^{j-1}, \bar{h}, p \right) \tag{6.34}
\]

and by substituting \( t_0 = a \left( \frac{1 - p (a)}{1 - p (a)^a} \right) \), we get that \( f_{i,j} (t) = f_i^{\text{SINGLE-B}} \left( 1, m, \alpha_j, \bar{h}, p \right) \).

Finally, as in the proof of Theorem 21, the MHT-B scheme cannot bring to any desired average number of bucket accesses \( a \), but is limited to \( f_{\text{MHT-B}}^a \left( 1 \right) \). Since \( f_{\text{MHT-B}}^a (t) \) increases as \( k_0 \) decreases, and \( k_0 \) decreases as \( \gamma \) increases, then the minimum \( \gamma \) such that the MHT-B scheme achieves optimality, for a given \( a \) and \( r \), is given by \( \gamma_0 \) such that \( f_{\text{MHT-B}}^a (1) = a \).
Since \( t_0 = a \left( \frac{1 - p (a)}{1 - p (a)^a} \right) \), we seek \( \gamma \) such that \( a = \frac{1 - p (a)^a}{1 - p (a)} \). Thus, we get the optimality region of this scheme. \( \square \)

6.10 Comparative Evaluation and Analysis

Fig. 6.4(a) shows the optimality region of SEQUENTIAL-B and MHT-B with element-per-bucket ratio \( r = 8 \), and \( d = 3 \) hash functions. For each value of the average number of bucket accesses \( a \), it shows the minimum value of the overflow fraction \( \gamma \) that suffices to solve the optimal access-constrained balancing problem. For instance, we can see that for \( a \approx 1.1 \), SEQUENTIAL-B achieves optimality for an overflow fraction equal to or larger than approximately 1%. In addition, Fig. 6.4(b) shows the optimality region of SEQUENTIAL-B and MHT-B with \( a = 1.2 \) and \( d = 3 \) for different values of \( r \). We can see that MHT-B scales better to higher loads.

6.11 Analysis of the Balanced Bloom Filter

In Section 6.3 we provided a high-level overview of the Balanced Bloom filter. In this section, we explain in detail how to use our optimal online schemes to construct the Balanced Bloom filter, and by this, to achieve a better FPR for networking devices.
6.11.1 Balanced Bloom Filter with Overflow List

*Blocked Bloom filters* [29, 30] form the first attempt to design an access-efficient Bloom filter. They constrain the $k$ hashed bits to be located in the same memory block, thus causing a single memory access.

The Blocked Bloom filter mechanism can be modeled using the single-B scheme with no overflow list ($\gamma = 0$) and with a cost function $\phi$ that expresses the FPR incurred to an element in a given memory block given the number of elements are hashed to this memory block:

$$
\phi(i) = \left(1 - \left(1 - \frac{1}{B}\right)^{ki}\right)^B \approx \left(1 - e^{-\frac{ki}{B}}\right)^k,
$$

(6.35)

where $B$ is the size in bits of the memory block and $k$ is the number of hash functions used. Although the single-B scheme is optimal, its average number of memory accesses $a$ is 1, thus it achieves poor balancing of the elements resulting in a high FPR. In this section, we explain in detail how to *use our optimal online schemes to achieve a better balancing between the memory blocks, and consequentially, a better FPR.*

Assuming memory blocks of size $B$ bits and bits-per-element ratio $\beta$, the number of elements per bucket $r$ is $B/\beta$. Using the optimal online balancing schemes described above to implement a Balanced Bloom filter requires saving $b = \lceil \log_2 (k_0 + 2) \rceil$ bits in every memory block.
block to count the elements hashed into each one. Thus, we get $\phi(i) = \left(1 - e^{-\frac{ki}{B}}\right)^k$.

In the standard Bloom filter, the optimal FPR is achieved when using $k = r \cdot \ln 2$ hash functions [13]. Although this may not be the best choice in our settings, we will use the same $k$ for simplicity.

Balancing schemes that read more than one memory block on a query operation increase the FPR, because a false positive may result from a query operation on each one of the memory blocks. Thus, since the probability of false positive in every memory block is relatively small, then the overall false positive probability, i.e. the FPR, is modeled by $\sum_{j=1}^{d} P_j \cdot \text{FPR}_j$, where $P_j$ is the probability that a query operation results in reading at least $j$ memory blocks, and $\text{FPR}_j$ is the false positive probability of the $j$-th memory block read. Since in our balancing schemes, there exists some constant FPR, where for all $j$, $\text{FPR}_j = \text{FPR}$, the expression above reduces to $\sum_{j=1}^{d} P_j \cdot \text{FPR} = \mathbb{E}(P) \cdot \text{FPR}$, where $\mathbb{E}(P)$ is the expected number of memory-block read operations.

Given an average number of memory accesses $a$ in our balancing schemes, $\mathbb{E}(P) \neq a$ in general. Querying the Balanced Bloom filter with elements that are in the set is expected to have exactly $a$ memory accesses on average. This is because the query memory access pattern follows exactly the insertion memory access pattern. However, for elements that are not in the set, the average number of memory accesses is expected to be $\mathbb{E}(P) \leq d$.

Fig. 6.5 compares the FPR for different values of bits-per-element ratios with memory block size $B = 256$. The MHT-B balancing scheme has $a = 1.2$ and $d = 3$. Thus, by Theorems 18 and 23, we get that the overflow list size needed is $\gamma \approx 0.5\%$. For comparison, Fig. 6.5 presents the performance of both the single-B balancing scheme with the same overflow list size and the Blocked Bloom filter, which is equivalent to the single-B scheme with no overflow list.

Because of the usage of the small overflow list, the single-B balancing scheme performs only slightly better than the Blocked Bloom filter scheme. For low values of bits-per-element ratio, the MHT-B scheme performs worse. This is due to the need to check multiple memory blocks (up to $d$) on a query operation. However, demonstrating the power in balancing, for
larger values of bits-per-element ratios, the MHT-B performs better by up to two orders of magnitude, and only one order of magnitude worse than the standard Bloom filter, which uses an access-inefficient scheme with \( a \approx k \) accesses. For example, for a bit-per-element ratio of 24, \( k = 17 \) hash functions are used, introducing up to 17 memory-read operations in the standard Bloom filter, which can clearly present memory-throughput and power-consumption issues.

6.11.2 Balanced Bloom Filter Without Overflow List

In some applications, it may be too expensive to use an overflow list. In such cases, the overflow list can be avoided using a simple modification on our basic scheme. Upon an insertion of a new element, the basic scheme runs as usual, but whenever there is an overflow event, the corresponding element is just put in one of the \( d \) choices at random. Because there is only a small fraction of overflow elements, this is not expected to worsen the FPR of the system significantly. However, in the general OPTIMAL ACCESS-CONSTRAINED BALANCING PROBLEM, we are no longer able to prove theoretically the optimality of the solution, as we could in case one may use an overflow list of a small size.

6.12 Trace-Driven Experiments

We conducted experiments of our proposed MHT-B-based Balanced Bloom filter using real-life traces recorded on a single direction of an OC192 backbone link [98], where packets are hashed to the memory blocks using a real 64-bit mix function [99]. Our goal is two-folded. First, we would like to verify that our analysis agrees with results of real-life traces. And second, we would like to evaluate the case where no overflow list is used, as introduced in Section 6.11.2.

In all experiments, we used the MHT-B-based Balanced Bloom filter with \( a = 1.2 \), \( d = 3 \), memory block sizes of either \( B = 256 \) or \( B = 512 \), a total of 1024 memory blocks, and various values of loads \( r \). By Theorems 18 and 23, we get that the overflow list size required in the basic scheme is \( \gamma \approx 0.5\% \). We further find by Theorem 23 the partition of the memory blocks to subtables. Notice that this partition depends on the load \( r \). To hash the elements into the memory blocks, we used a real 64-bit mix function. However, to set the \( k \) bits within the corresponding memory block, we used a standard randomization procedure. The total FPR is then computed by the number of set bits and the number of elements recorded by the counter at each memory block.

Fig. 6.6 shows the FPR found by our experiments of both the basic MHT-B-based Balanced Bloom filter scheme, and the variation where no overflow list is used. First, the experiment results of the basic MHT-B-based scheme verifies the accuracy of our analysis. We note that our experiments also show that the fraction of elements in the overflow list and the average number of access operations per element are approximately \( \gamma = 0.5\% \) and \( a = 1.2 \), respectively. These results further validate our analysis. In addition, if an overflow
For a bits-per-element ratio of 40 and a memory block size $B = 256$, our experiments show an FPR of $2.0 \cdot 10^{-7}$ and $3.6 \cdot 10^{-7}$ for the basic scheme and the scheme with no overflow list, respectively. While for the same configuration with a memory block size $B = 512$, experiments show FPRs of $3.7 \cdot 10^{-8}$ and $1.1 \cdot 10^{-7}$.

Finally, Fig. 6.6 also depicts the FPR of the blocked Bloom filter [29, 30], and shows that even if no overflow list is used, the FPR can be drastically reduced: For instance, for a bits-per-element ratio of 40 and a memory block size $B = 256$, our experiments show that the FPR is reduced from $3.4 \cdot 10^{-5}$ to $3.6 \cdot 10^{-7}$. This improvement of approximately two orders of magnitude demonstrates the power of the suggested balancing schemes.

6.13 Additional Application: Variance of the Query-Time in Chain-Based Hash-Tables

The balancing problem can be directly used in order to construct an access-constrained balancing scheme with optimal variance over its query time. In such a scheme, the time it takes to complete a lookup operation directly depends on the occupancy of the buckets. For example, suppose that there is only one hash-function $H$, and some element $y$ is mapped
to a bucket $H(y)$; in order to query $y$ we need to go over all elements of $H(y)$ in case $y$ is not in the table, or on average over half of them in case $y$ is in the hash-table.

Note that the average load is simple to obtain and equals $\frac{(1-\gamma)n}{m}$. We next show how to find balancing schemes with minimal occupancy variance. This is simply done by defining the appropriate cost function: $\phi(i) = i^2 - \left(\frac{(1-\gamma)n}{m}\right)^2$.

Thus,

$$
\phi_{\text{HAL}} = \lim_{m \to \infty} \frac{1}{m} \sum_{j=1}^{m} \mathbb{E}\left(O_j^2 - \left(\frac{(1-\gamma)n}{m}\right)^2\right) = \lim_{m \to \infty} \frac{1}{m} \sum_{j=1}^{m} \mathbb{E}(O_j^2) - \mathbb{E}(O_j)^2
$$

$$
= \lim_{m \to \infty} \frac{1}{m} \sum_{j=1}^{m} \text{Var}(O_j) = \text{Var}(O),
$$

(6.36)

where $O_j$ is the random variable representing the occupancy of bucket $j$. By symmetry, all variables $O_j$ have the same distribution and thus the same variance, which is denoted by $\text{Var}(O)$. This immediately implies that the schemes we presented can be used in order to build a hash-table with optimal variance.
Chapter 7

Tables in Combined Memories

In some cases, hash tables often require a combined memory model, in which some of the elements are stored in a fast memory (for example, cache or on-chip SRAM) while others are stored in much slower memory (namely, the main memory or off-chip DRAM). This makes the implementation of real-life hash tables particularly delicate, as a suboptimal choice of the hashing scheme parameters may result in a higher average query time, and therefore in a lower throughput. In network devices, hash tables are often on the critical packet path, thus affecting the overall throughput of the device.

In this chapter, we focus on multiple-choice hash tables. Given the number of choices, we study the tradeoff between the load of a hash table and its average lookup time. The problem is solved by analyzing an equivalent problem: the expected maximum matching size of a random bipartite graph with a fixed left-side vertex degree. Given two choices, we provide exact results for any finite system, and also deduce asymptotic results as the fast memory size increases. In addition, we further consider other variants of this problem and model the impact of several parameters. Finally, we evaluate the performance of our models on Internet backbone traces, and illustrate the impact of the memories speed difference on the choice of parameters. In particular, we show that the common intuition of entirely avoiding slow memory accesses by using highly efficient schemes (namely, with many fast-memory choices) is not always optimal.

Unlike typical hash tables, network hash tables have two specificities. First, they are rebuilt infrequently. For all practical purposes, we can assume that they are built offline. Second, they need to process elements with query/modify requests extremely fast, using a small and bounded number of memory accesses. For example, a network hash table may store the states of a given number of flows, or the bills of a given number of customers. The set of flows or customers in the hash table is assumed to be predetermined. However, at each new packet arrival, the hash table needs to be accessed immediately and within a bounded time. Thus, multiple-choice hashing scheme are are particularly suitable to network hash tables [22, 23]. In these schemes, each element can only be stored in one of \(d\) possible fixed-size buckets, usually of size 1. Consequentially, to perform a lookup
operation we would need to check each one of the $d$ corresponding buckets.

In a typical setting, a network hash table needs to support $n$ elements using an SRAM size of $m$ buckets. Given that it relies on multiple-choice hashing, each of the $n$ elements can hash into $d$ arbitrary buckets using independent hash functions. Then, the network hash table designer faces several fundamental tradeoffs. For example, if $d$ is too small, i.e. each element can only hash into a few buckets, the hashing scheme may not be efficient. Therefore, more elements may need to be placed in the slow DRAM. On the other hand, if $d$ is too large, even if all elements are stored in the SRAM, it may take too long to check in which bucket (out of the $d$ potential SRAM buckets) each element actually resides.

As a result, incorrect hash-table settings can significantly increase the average delay needed to deal with each packet. Therefore, when incoming packets are processed sequentially, incorrect settings can also significantly decrease the throughput of the network hash table. Naturally, to maximize the throughput of the hash table given the number of hash functions $d$, we should store the largest possible number of elements in the available SRAM memory. Since lookup operations are typically the vast majority of operations in network hash tables, we focus on the case of $d = 2$. This ensures a fast lookup operation, when the element is indeed in the SRAM.

In order to tackle this problem for any $d$, we consider the bipartite graph formed by the $n$ elements on one side, the $m$ (SRAM) buckets on the other, and $d$ links leaving each element for the buckets according to the hash values of the element. Naturally, to maximize the throughput of the hash table given the number of hash functions $d$, we should store the largest possible number of elements in the available SRAM memory. Since lookup operations are typically the vast majority of operations in network hash tables, we focus on the case of $d = 2$. This ensures a fast lookup operation, when the element is indeed in the SRAM.

Finally, although we are mainly interested in lookup operations in network devices, updates (i.e., insertion of new elements) can be made using a variant of cuckoo hashing [46]. Upon arrival of a new element, it is placed according to one of its $d$ hash values. If all buckets are full, it displaces another element, which is then moved to one of its other $d - 1$ buckets and so on. If after some predetermined number of element displacements no room is found, the element that was last displaced is stored in the slower memory. Since this algorithm is analogous to the seminal Ford–Fulkerson algorithm [110, 111] of finding augmenting paths in graphs, if the number of allowed displacements is high enough, it practically computes a maximum size matching on the new graph, thus achieving the maximum capacity of the hash table.

### 7.1 Summary of Our Results

Our first contribution is that we study the best possible performance of multiple-choice hashing schemes with $d = 2$. While it has been shown that there is multiple-choice hashing...
scheme (namely, cuckoo hashing) for which all elements could fit in the hash table with high probability up to a load $n/m = 0.5$ [39, 35, 38, 37], we also analyze the best possible performance when the load gets beyond 0.5. To do so, we essentially transform the problem into the above-mentioned graph theory problem, then provide a theoretical analysis, and later evaluate the real-life behavior by using Internet backbone traces.

Specifically, we study the expected maximum matching size in a random bipartite graph where the destinations of the $d$ outgoing edges from each left-side vertex are chosen uniformly at random. This models typical independent perfect hash functions, that often yields an excellent approximation of real-life hash functions [22, 112]. We decompose each random bipartite graph into connected components, and then separately analyze each component and evaluate the size of its local maximum bipartite match. Then, we count the number of connected components in the graph and thus derive the size of the maximum matching in the entire graph. Remarkably, we can obtain an exact expression of the average best possible performance in any finite system. This non-asymptotic analysis is particularly needed when $n$ and $m$ are known to be small, such as in cache architecture. We further show that the actual maximum matching size is sharply concentrated around its expected value. Thus, the difference between $n$ and the expected maximum matching size provides, with high probability, the number of elements that should be stored in the off-chip DRAM.

Second, we provide an exact analysis of a common multiple-choice hashing implementation in which the memory is (statically) partitioned into two segments, such that each segment corresponds to the image of one hash function; this implementation is particularly attractive when using single-ported memories.

Third, we present exact analysis when, in order to minimize the number of memory accesses, the average number of choices is less than 2.

Fourth, we obtain a lower bound on the required DRAM size when the number of hashes $d$ exceeds 2.

We further evaluate our results on real-life Internet packet traces from an OC192 backbone link, using a real-life 64-bit mix hash function. We show that when the load is 1, i.e. $n = m$, we can insert an average of 83.81% of the packets within the hash table. Likewise, when the load is 0.6, i.e. $n = 0.6m$, we can insert in average 99.38% of the packets, thus only storing 0.62% on off-chip DRAM. We further confirm our analytical models and show that our bounds for $d > 2$ are typically within 1% of the exact value.

Finally, we compare the network hash table throughput using different numbers $d$ of hash functions. We first provide analytical results when the on-chip memory is partitioned into two (unequal) segments. Then, we run simulations and show that, unlike common belief, it is still worth using $d = 2$ hash functions beyond a load of 0.5, even though some of the packets are stored on DRAM. We also illustrate that the exact load at which a system with $d = 3$ outperforms a system with $d = 2$ depends on ratio between the SRAM and the DRAM access times.
**Chapter Organization**  We start by introducing the preliminary definitions in Section 7.2. Then, Section 7.3 provides the expected maximum matching size of random bipartite graphs with left-side vertex degree 2, where a variation of the problem in which each left-side vertex degree is at most 2 is considered in Section 7.4. Next, in Section 7.5, we solve the more appealing problem in which the right-side vertices are partitioned into two subsets, and each left-side vertex has exactly one edge to each of these subsets. Section 7.6 provides an upper bound on the expected maximum matching size when the constant left-side vertex degree is at least three. Last, in Section 7.7 we verify and evaluate our results, including by real-life trace-based experiments.

### 7.2 Bipartite Graph Model

#### 7.2.1 Model

In this section, we define multiple-choice hashing using a bipartite graph, with the left-side vertices corresponding to elements and the right-side vertices to SRAM buckets.

Formally, given two disjoint sets of vertices \( L \) and \( R \) of size \( n \) and \( m \) respectively, we consider a random bipartite graph \( G = \langle L + R, E \rangle \), where each vertex \( v \in L \) has \( d \) outgoing edges whose destinations are chosen independently and uniformly at random among all vertices in \( R \). We allow both choices to be the same vertex, implying that \( G \) might have parallel edges. For brevity, we sometimes say that \( v \in L \) chooses a vertex \( v' \in R \) if \((v,v')\) is in \( E \). The load of \( G \) is denoted by \( c = \frac{n}{m} \).

We also consider a static partitioning of two choices; the set \( R \) is partitioned into two disjoint sets \( R_u \) and \( R_d \) of sizes \( \alpha_1 \cdot m \) and \( \alpha_2 \cdot m = (1 - \alpha_1) m \) (since \( \alpha_1 + \alpha_2 = 1 \)). In that case, we consider a random bipartite graph \( G_{\alpha_1} = \langle L + (R_u \cup R_d), E \rangle \), where each vertex \( v \in L \) chooses independently and uniformly at random exactly one vertex in \( R_u \) and another vertex in \( R_d \).

We want to find both the expected maximum matching size as well as the normalized limit expected maximum matching size for the above-mentioned graph models. To do so, we model our hash functions as fully random, which often yields an excellent approximation \([22, 112]\).

**Definition 10**  For any graph \( G \), let \( \mu(G) \) be the expected size of the maximum size matching.

Notice that if \( G \) is a deterministic graph, then \( \mu(G) \) is simply the size of its maximum size matching.

**Definition 11**  The normalized limit expected maximum matching size \( \psi = \lim_{n \to \infty} \frac{\mu(G)}{n} \) is the limit percentage of the expected maximum matching size (out of the number of the vertices in \( L \)).
Note that often we are interested in $n - \mu(G)$, which corresponds to the expected number of unmatched left-side vertices in the graph. This corresponds to the number of elements that should be stored in the off-chip DRAM.

Finally, our goal is to model the throughput of the network hash table. To do so, we first assume that each access to on-chip SRAM takes a unit amount of time, while each access to off-chip DRAM takes a latency of $b$ (where $b > 1$, e.g., $b = 10$). Also, all accesses are sequential. For instance, assume that we use with $d = 2$, and a given element is in the DRAM. Then a query for this element would first successively check the $d = 2$ SRAM buckets, then the DRAM, for a total latency of $b + 2$.

We further assume that all the elements in the hash table are equally likely to be accessed. We define the average latency as the average total access latency over all elements in the hash table, including the elements in the SRAM as well as in the DRAM. We further define the throughput of the network hash table as the inverse of its average latency. For example, if it takes on average 2 accesses to query an element, then the hash table throughput is $\frac{1}{2}$. Our goal is to maximize this throughput.

### 7.2.2 Assumptions

We make several key assumptions that may limit the reach of our results.

First, we assume that the network hash table is built offline. So we can store a number of elements in the SRAM as large as the maximum matching size. Furthermore, we assume that the $d$ buckets can only be accessed serially. Therefore, although no multiple-choice algorithm with $d = 2$ can beat the above SRAM utilization, it may theoretically yield a better throughput. This is because it may be better to store less elements in the SRAM, most of them in their first choice. This may improve the overall delay although more elements need to be stored in the DRAM.

Our second assumption is that the hash table is accessed sequentially, such that each new packet needs to wait for the end of the former packet. As a result, throughput is inversely proportional to packet delay. This assumption is designed to cope with general hash tables, in which several applications may share the hash table, and therefore each packet may need to access and modify several elements in the hash table according to different application-based keys. Since the modifications of each packet may also affect the next packets, it is simpler to wait for its processing to end. The hash table may be made more efficient by processing packets of different application-based flows in parallel. But such a scheme may become too hard to implement for a large number of applications, because each key of each packet needs to be compared with the relevant keys of all previous packets currently accessing the hash table. In any case, our results can also be extended to such parallel accesses.

Our third assumption is that all element queries are for elements that are indeed stored in the hash-table. This assumption is common in several networking applications [22], while in others it requires a set membership query before actually accessing the hash-table. To
obtain the expected latency, we further assume that all elements in the hash table are equally likely to be accessed.

Finally, our last assumption is that each access to DRAM is \( b \) times slower than an access to SRAM, i.e. the impact of DRAM is mainly through its access time. We do not take into account the chip in/out pin capacity, which may further reduce the range of hashing options available. As a first approximation, we also do not consider the DRAM division into banks, and do not consider non-uniform DRAM access times.

### 7.3 Bipartite Graphs with \( d = 2 \)

We are now interested in evaluating the expected best performance of multiple-choice hashing schemes with \( d = 2 \). As explained above, we approach the problem using a graph-theory perspective, since it is the same as evaluating the expected maximum matching size of the random bipartite graph \( G \).

To do so, we consider the connected components of the random bipartite graph \( G \). We start by stating some lemmas on these connected components, before establishing our main result on the expected matching size. Note that further evaluation of the results reported here appears in Section 7.7.

#### 7.3.1 Expected Maximum Matching Size

We first consider an arbitrary bipartite graph \( H = (L_H + R_H, E_H) \), where each left-side vertex in \( L_H \) chooses \( d = 2 \) right-side vertices in \( R_H \) (parallel edges are allowed), with \( |L_H| = s \) and \( |R_H| = q \).

Figure 7.1 illustrates such a bipartite graph with \( s = 3 \), \( q = 4 \), and left-side vertex degree 2. Dashed lines represent edges not in the maximum size matching, while solid lines represent edges in the maximum size matching.

We start by quoting a few useful and straightforward lemmas, before stating our result.

Figure 7.1: Example of bipartite graph with left-side vertex degree 2
Lemma 2 If $s \leq q - 2$, then $H$ is not connected.

Proof. The proof follows by induction on $s$. For $s = 1$, there are 2 edges in the graph and therefore every graph with $q \geq 3$ is not connected. Assume that the claim holds up until $s = s'$, we next prove that it holds for any bipartite graph $H'$ such that $|L_{H'}| = s' + 1$ and $|R_{H'}| \geq s' + 3$. Assume towards a contradiction that there is a graph $H'$ that is connected. We first show that there is a vertex in $R_{H'}$ with a degree 1: This follows from the fact that the average right-side degree is $\frac{2(s' + 1)}{s' + 3} < 2$, implying that there is at least one vertex with degree strictly less than 2; since the graph is connected, there are no right-side vertices with degree 0. Let $v_r$ be such a vertex and let $v_r \in L_{H'}$ be the (only) left-side vertex to which it is connected. By the induction hypothesis, the graph induced by $L_{H'} \{v_r\}$ and $R_{H'} \{v_r\}$ is not connected, implying it has at least two connected components. In $H'$, $v_r$ is connected to $v_r$ and since its degree is 2 it can be connected only to one of these components. This implies that $H'$ is also not connected, and the claim follows.

Lemma 3 If $H$ is connected and $s \geq q$, then $\mu(H) = q$.

Proof. We first consider the case where $s = q$. For $S \subseteq L_H$, let $d(S) \subseteq R_H$ be the set of vertices that are adjacent to any vertex in $S$. Hall’s Theorem [113] implies that to prove that $\mu(H) = q$ (namely, there is a perfect matching in $H$) it suffices to prove that for every $S \subseteq L_H$, $|S| \leq |d(S)|$. Assume towards a contradiction that there is a subset $S \subseteq L_H$ such that $|S| > |d(S)|$, and denote $|d(S)|$ as $\ell$. Furthermore, consider the bipartite graph $\hat{H} = \langle \hat{L}_H + \hat{R}_H, \hat{E}_H \rangle$, in which $\hat{L}_H = L_H \setminus S$, $\hat{R}_H = R_H \cup \{\hat{v}_R\} \setminus d(S)$ (where $\hat{v}_R$ is a newly-introduced vertex) and any edge in $E(H)$ of the form $(v_\ell, v_r)$ such that $v_\ell \in L_H \setminus S$ and $v_r \in d(S)$ is replaced with the edge $(v_\ell, \hat{v}_R)$ in $\hat{E}_H$. Notice that since $H$ is connected, $\hat{H}$ must be connected as well. Recall that $|S| > \ell$, thus $|\hat{L}_H| = |L_H \setminus S| \leq s - \ell - 1$, while $|\hat{R}_H| = |R_H \cup \{\hat{v}_R\} \setminus d(S)| = |R_H| - |d(S)| + 1 = s - b + 1$. This contradicts Lemma 2, implying that for every $S \subseteq L_H$, $|S| \leq |d(S)|$ and by Hall’s Theorem $\mu(H) = q$.

For $s > q$, trivially $\mu(H) \leq q$. Therefore, it suffices to show that there exists a subset $S \subseteq L_H$ of size $q$, such that the corresponding bipartite subgraph is connected (and hence has a perfect matching of size $q$). We construct $S$ in $q$ iterations such that at the end of iteration $n$ we end up with some subsets $S_n \subseteq L_H$ and $Q_n \subseteq R_H$ of the same size $n$, whose corresponding subgraph is connected. We start by $n = 1$ and pick some vertex $v_{R_1} \in R_H$ and one of its adjacent vertices $v_{L_1} \in L_H$. Assuming that at the end of iteration $n$, sets $S_n$ and $Q_n$ were chosen (and their corresponding graph is connected), we next construct $S_{n+1}$ and $Q_{n+1}$. Let $v_1$ be an arbitrary vertex in $S_n$ and let $v_2$ be an arbitrary vertex in $L_H \setminus S_n$ (such a vertex always exists since $s > q > n$). Similarly, let $v_1'$ be an arbitrary vertex in $Q_n$ and let $v_2'$ be an arbitrary vertex in $R_H \setminus Q_n$. Since $H$ is connected there is a path between $v_1$ and $v_2$, and let $v$ be the first vertex along this path that is not in $S_n$. Similarly, $v'$ is the first vertex along the path between $v_1'$ and $v_2'$ that is not in $Q_n$. We differentiate between three cases: (i) $v$ is adjacent to $Q_n$ and $v'$ to $S_n$. In this case
\[ S_{n+1} = S_n \cup \{v\} \] and \( Q_{n+1} = Q_n \cup \{v'\} \) and the corresponding subgraph is connected; (ii) \( v \) is not adjacent to a \( Q_n \). Let \( w \) be the vertex before \( v \) in the path between \( v_1 \) and \( v_2 \), and let \( w' \) be the vertex before \( w \) in the path. Note that \( w' \in S_n \) by the choice of \( v \), and that \( w \notin Q_n \) (otherwise \( v \) is adjacent to a \( Q_n \)). Thus, for \( S_{n+1} = S_n \cup \{v\} \) and \( Q_{n+1} = Q_n \cup \{w\} \), the corresponding subgraph is connected; (iii) \( v' \) is not adjacent to a \( S_n \). The claim holds similarly to case (ii) by looking at the path between \( v'_1 \) and \( v'_2 \). We continue this construction for \( q \) iterations, resulting in two subsets \( S_q \subseteq L_H \) and \( Q_q \subseteq R_H \) of size \( q \) each, whose corresponding subgraph is connected.

**Lemma 4** If \( H \) is connected and \( s = q - 1 \) then \( \mu(H) = s \).

**Proof.** Since each vertex in \( L_H \) has a degree of two, the sum of the degrees of all the vertices in \( R_H \) is \( 2s = 2q - 2 \). Therefore, there must be at least one vertex \( v_r \in R_H \) with degree 1 (there cannot be a vertex with degree 0 since \( H \) is connected). Let \( v_L \in L_H \) be the (only) vertex that is connected to \( v_R \) and \( \hat{v}_R \in R_H \) be the other vertex that is connected to \( v_L \). Also consider the bipartite graph \( \hat{H} = (\hat{L}_H + \hat{R}_H, \hat{E}_H) \) that is given by removing \( v_R \) from \( H \) and adding a new edge \( (v_L, \hat{v}_R) \). By the construction of \( \hat{H} \), the degree of each vertex in \( \hat{L}_H \) is exactly 2. Moreover, since \( H \) is connected, \( \hat{H} \) is also connected. Hence, Lemma 3 implies that there is a matching of size \( s \) in \( \hat{H} \). By the construction of \( \hat{H} \), this is also a matching in graph \( H \).

**Lemma 5** For any graph with \( s = q - 1 \), \( H \) is connected if and only if it is a tree.

**Proof.** First, if \( H \) is a tree then it is connected by definition. To show the other direction, we assume towards a contradiction that \( H \) is a connected graph with cycles; let \( C \) be a cycle in \( H \), and consider an edge \( e = (v_L, v_R) \) that resides at cycle \( C \) (where \( v_L \in L_H \) and \( v_R \in R_H \)). We build the bipartite graph \( \hat{H} = (\hat{L}_H + \hat{R}_H, \hat{E}_H) \), such that \( \hat{L}_H = L_H \), \( \hat{R}_H = R_H \cup \{\hat{v}_R\} \), where \( \hat{v}_R \) is a newly-introduced vertex, and \( \hat{E}_H = E_H \setminus \{e\} \cup \{\hat{e}\} \), where \( \hat{e} = (v_L, \hat{v}_R) \). Intuitively, we replace one of the edges in the cycle to reach for a newly-introduced vertex, and by that we increase the size of the connected component. Notice that \( \hat{H} \) is connected and all vertices in \( \hat{L}_H \) have a degree of 2. But, \( |\hat{L}_H| < |\hat{R}_H| - 1 \), thus contradicting Lemma 2 and the claim follows.

**Lemma 6** The number \( T_s \) of labeled connected bipartite graphs \( H \) whose \( |L_H| = s \) and \( |R_H| = s + 1 \) is \( T_s = (s + 1)^{s-1} \).

**Proof.** We count the connected bipartite graphs with two disjoint sets \( L_H \) and \( R_H \). By Lemma 5, we have to count the number of trees over the set \( L_H \cup R_H \), where edges must be of the form \((v_L, v_R)\), such that \( v_L \in L_H \) and \( v_R \in R_H \). We build (and count) the set as follows: The number of trees over the set \( R_H \) is \((s+1)^{s-1}\) (Cayley’s formula). For each such tree instance, we put a new vertex (originally from \( L_H \)) between each pair of adjacent vertices. There are \( s! \) possibilities to do so.
We can now prove the next theorem on our random bipartite graph $G$, which is the main theoretical result of this chapter. This theorem provides the exact expected number of elements $\mu(G)$ that can fit the on-chip SRAM memory with $d = 2$. Therefore, a corollary is that $n - \mu(G)$ also gives us the expected number of elements that are left outside the chip.

**Theorem 24** Let $d = 2$ and $\ell = \min(n, m - 1)$. The expected maximum matching size $\mu(G)$ is

$$\mu(G) = m - \sum_{s=0}^{\ell} \left\{ \binom{n}{s} \cdot \binom{m}{s+1} \cdot \left(1 - \frac{s+1}{m}\right)^{2(n-s)} \cdot \left(\frac{s+1}{m}\right)^{2s} \cdot \frac{2^s s!}{(s+1)^{s+1}} \right\}.$$

**Proof.** Let $M$ be a maximum matching of $G$. Our proof is based on counting the expected number of vertices in $R$ that are not part of $M$, and on the decomposition of $G$ into its connected components.

Lemma 2 yields that any connected component of $G$ with $s$ left-side vertices has at most $s + 1$ right-side vertices. We call a connected component with $s$ left-side vertices and $s + 1$ right-side vertices a deficit component of size $s$. Lemma 4 implies that the maximum matching size of any such deficit component is $s$. Therefore, exactly one of its right-side vertices is not part of $M$. Notice that in all other connected components, where $q < s + 1$, the maximum matching size of $G$ is exactly $q$ (Lemma 3), implying that all their right-side vertices are part of $M$.

Thus, in order to calculate the size of $M$, it suffices to count the number of deficit components $x$. The size of $M$ is $m - x$ because exactly $x$ right-side vertices do not participate in $M$, one for each deficit component.

Consider a random bipartite graph, with $s$ left vertices, each of degree 2, and $s + 1$ right vertices, and let $P_s = \frac{2^s T_s}{(s+1)^s}$ be the probability that it is connected. Note that we multiply $T_s$ by $2^s$ because $T_s$ only counts connected bipartite graphs, which are necessarily trees (Lemma 5), with no distinction between the two edges connected to each left vertex, while in the denominator we count all possible instances of random bipartite graphs as above, where we distinguish between the two edges connected to each left vertex.

The expected number of deficit components of size $s$ is $\binom{n}{s} \binom{m}{s+1} \cdot \left(1 - \frac{s+1}{m}\right)^{2(n-s)} \cdot \left(\frac{s+1}{m}\right)^{2s} \cdot \frac{2^s s!}{(s+1)^{s+1}}$. $P_s$. The above expression consists of the following factors (in order):

(i) choosing the $s$ vertices in $L$;
(ii) choosing the $s + 1$ vertices in $R$;
(iii) the probability that all $s + 1$ vertices in $R$ may be connected only to the chosen $s$ vertices in $L$;
(iv) the probability that all $s$ vertices in $L$ are only connected to the $s + 1$ vertices in the right side; and,
(v) the probability that all chosen vertices are connected.
Finally, we calculate $x$ by summing over all possible values on $s$. As mentioned before, the expected size of $M$ is given by $m - x$. We get: $\mu(G) = m - \sum_{s=0}^{\ell} \binom{s}{k} \binom{m}{s+1} \cdot (1 - \frac{s+1}{m})^{2(n-s)} \cdot \binom{s+1}{m} \cdot P_s$, where $\ell = \min(n, m - 1)$, $P_s = \frac{2^{s+1}}{(s+1)^2}$, and $T_s = (s+1)^{s-2} \cdot (s+1)!$, as found in Lemma 6.

The following example provides a simple illustration when $n = m = 2$.

Example 11 Consider the case $n = m = 2$ (and $d = 2$). Then in all random graphs the maximum matching size is 2, except for the two extreme cases where all 4 edges are connected to a specific vertex in $R$, and then the maximum matching size is 1. Each such case occurs with probability $\left(\frac{1}{2}\right)^4$. Hence, $\mu(G) = 2 - 2 \cdot \left(\frac{1}{2}\right)^4 = \frac{15}{8} = 1.875$. This is indeed precisely what we obtain using Theorem 24.

7.3.2 Concentration Result

We next show that the size of the maximum matching is highly concentrated around its expectation $\mu(G)$. This implies that the number of off-chip elements will be close to its average value.

In order to prove this result, we apply Azuma’s inequality to a Doob martingale (more specifically, the martingale is a vertex exposure martingale of the left-side vertices). Note that as long as all left-side vertices pick their edges independently, this concentration result holds regardless of the value of $d$, and more generally regardless of the specific distribution over which the hash functions are defined. Therefore, the concentration result also applies to the settings of the next sections.

Theorem 25 Let $H$ be a specific instance of the random graph $G$, as defined in Section 7.2. For any $\lambda > 0$, $\Pr(|\mu(H) - \mu(G)| > \lambda \sqrt{n}) < 2e^{-\lambda^2/2}$.

Proof. Our notations follow those of [114]. We first define an exposure martingale, which exposes one left-side vertex at a time, along with all its outgoing edges. This martingale is equivalent to a regular vertex exposure martingale, in which all right-side vertices are exposed first, and then left-side vertices are exposed one by one.

Specifically, let $G$ be the probability space of all two-choice bipartite graphs as defined in Section 7.2 and $f$ the size of the maximum size matching of a specific instance. Assume an arbitrary order of the left-side vertices $L = \{v_1, \ldots, v_n\}$, and define $X_0, \ldots, X_n$ by $X_i(H) = E[f(G) \mid \forall x \leq i, \forall v_y \in R, (v_x, v_y) \in G \text{ iff } (v_x, v_y) \in H]$. Note that $X_{\ell}(H) = \mu(G)$ since no edges were exposed, while $X_n(H) = \mu(H)$ as all edges are exposed.

Clearly, $f$ satisfies the vertex Lipschitz condition since if two graphs $H$ and $H'$ differ at only one left-side vertex, $|f(H) - f(H')| \leq 1$ (either that vertex is in the maximum matching or not). Thus, since each left-side vertex makes independent choices, [114, Theorem 7.2.3] implies that the corresponding vertex exposure martingale satisfies $|X_{i+1} - X_i| \leq 1$. Hence, by applying Azuma’s inequality, we immediately get the concentration result. \qed
Notice that if we are interested only in one-sided bounds, we can get a slightly tighter result: \( \Pr(\mu(G) - \mu(H) > \lambda \sqrt{n}) < e^{-\lambda^2/2} \). This is exploited in the following corollary, which shows that to obtain a given overflow fraction, the number of off-chip elements grows sub-linearly with \( n \) beyond its average value.

**Corollary 26** With probability at least \( 1 - \epsilon \), the number of elements that need to be stored in off-chip DRAM is less than \( n - \mu(G) + \sqrt{2n \cdot \ln(1/\epsilon)} \), where \( \mu(G) \) is as in Theorem 24.

**Proof.** If a stash of size \( n - \mu(G) + \sqrt{2n \cdot \ln(1/\epsilon)} \) is used, any hashing scheme fails if and only if \( n - \mu(H) > n - \mu(G) + \sqrt{2n \cdot \ln(1/\epsilon)} \), or by rewriting it, \( \mu(G) - \mu(H) > \sqrt{2n \cdot \ln(1/\epsilon)} \).

By substituting \( \lambda = \sqrt{2 \cdot \ln(1/\epsilon)} \) in the above one-sided bound, we get the claimed result. \( \square \)

### 7.3.3 Limit Normalized Expected Maximum Matching Size

Our results above provide exact expressions, given \( n \) elements and \( m \) SRAM buckets. We now want to study the scaling properties of the hash table, and are interested in the asymptotic expression where \( n \to \infty \) with \( c = \frac{n}{m} \) constant. To do this, we compute the limit of \( \frac{\mu(G)}{n} \) as \( n \to \infty \) such that \( c = \frac{n}{m} \). It results in an interesting connection between the limit normalized expected maximum matching size and the Lambert-W function, and even a connection between the perfect matching threshold and the radius of convergence of the Lambert-W function [97].

Before we continue with our results we first describe the Lambert-W function. This function, usually denoted by \( W(\cdot) \), is given by the following implicit representation:

\[
z = W(z) \cdot e^{W(z)},
\]

where \( z \) is a complex number [97].

For real valued arguments, i.e., \( z \) is real valued, \( W(z) \) has two real-valued branches: the principal branch, denoted by \( W_0(\cdot) \) and the branch \( W_{-1}(\cdot) \). Figure 7.2 shows the two real-valued branches. For instance, \( W_0(-e^{-1}) = W_{-1}(-e^{-1}) = -1 \) and \( W_0(0) = 0 \).

Note that the notation \( W(\cdot) \) usually relates to the principle branch, i.e. \( W_0(\cdot) \). Thus, although one would expect that for real-valued \( z \), \( W(z \cdot e^z) = z \), this is only the case for \( z \geq -1 \); in case \( z < -1 \), \( W_{-1}(z \cdot e^z) = z \neq W(z \cdot e^z) \).

We now give our result of this section.

**Theorem 27** Let \( d = 2 \). The limit normalized expected maximum matching size \( \psi = \lim_{n \to \infty} \frac{\mu(G)}{n} \) is given by:

\[
\psi = \frac{1}{c} + \frac{1}{2c^2} \cdot W(-2c \cdot e^{-2c}) + \frac{1}{4c^2} W^2(-2c \cdot e^{-2c}),
\]

where the Lambert-W function is the inverse function of the function \( \omega(x) = xe^x \).
Proof. We compute the limit of $\frac{\mu(G)}{n}$ as $n \to \infty$ such that $c = \frac{n}{m}$.

\[
\psi = \lim_{n \to \infty} \frac{1}{n} \cdot \left( m - \sum_{s=0}^{\ell} \binom{n}{s} \cdot \binom{m}{s+1} \cdot \left( 1 - \frac{s+1}{m} \right)^{2(n-s)} \cdot \left( \frac{s+1}{m} \right)^{2s} \cdot \frac{2^s s!}{(s+1)^{s+1}} \right)
\]

We find through differentiation that $(1 - \frac{s+1}{m})^{2(n-s)}$ is an increasing function with respect to $n$ (where $m = \frac{n}{c}$). Moreover, the expansion of $\frac{1}{n} \cdot \binom{n}{s} \cdot \frac{m}{s+1} \cdot (\frac{s+1}{m})^{2s}$ shows that it is also an increasing function. Therefore, their product is also increasing and, by the monotone convergence theorem [115], we get

\[
\psi = \lim_{n \to \infty} \frac{m}{n} - \sum_{s=0}^{\infty} \lim_{n \to \infty} \left( \frac{1}{n} \cdot \binom{n}{s} \cdot \binom{m}{s+1} \cdot \left( 1 - \frac{s+1}{m} \right)^{2(n-s)} \cdot \left( \frac{s+1}{m} \right)^{2s} \cdot P_s \right)
\]

where by convention $\binom{u}{v} = 0$ for $u < v$. By substituting the expression for $P_s$, and using the facts that $\binom{n}{s} = \frac{n^s}{s!} + O(n^{s-1})$ and $\lim_{n \to \infty} (1 + a/n)^n = e^a$, we deduce:

\[
\psi = \lim_{n \to \infty} \frac{m}{n} - \sum_{s=0}^{\infty} \lim_{n \to \infty} \left( \frac{1}{n} \cdot \frac{m^{s+1}}{s!} \cdot e^{-2\epsilon(s+1)} \cdot \frac{(s+1)^{2s}}{m^{2s}} \cdot \frac{2^s (s+1)^{s-1} s!}{(s+1)^{2s}} \right)
\]

111
By substituting \( m = \frac{n}{c} \), and simplifying the above expression, we get:

\[
\psi = \frac{1}{c} - \frac{1}{c} \sum_{s=0}^{\infty} e^{s} \cdot 2^{s} \cdot \frac{(s+1)^{s-1}}{(s+1)!} \cdot e^{-2c(s+1)}
\]

\[
= \frac{1}{c} - \frac{1}{2c^2} \sum_{s=1}^{\infty} (-2c \cdot e^{-2c})^{s} \cdot \frac{(-j)^{s-2}}{j!}
\]

Let \( T(x) = \sum_{j=1}^{\infty} \frac{(-j)^{j-2}}{j!} \cdot x^j \) be a formal power series, where by substituting \( x = -2c \cdot e^{-2c} \) we get the above expression. By differentiating \( T(x) \) and multiplying by \( x \), we get:

\[
x \cdot \frac{d}{dx} T(x) = - \sum_{j=1}^{\infty} \frac{(-j)^{j-1}}{j!} \cdot x^j = -W(x),
\]

where the Lambert-W function is the inverse function of the function \( \omega(x) = xe^x \) [97], and the last equality follows from its known Taylor expansion that converges as long as \( x \) is within the radius of convergence with \( |x| \leq e^{-1} \) [97].

Given that \( x \cdot \frac{d}{dx} T(x) = -W(x) \), we compute \( T(x) \):

\[
T(x) = \int \frac{1}{x} \cdot (-W(x)) \, dx = -W(x) - \frac{1}{2} W^2(x),
\]

with convergence within \( |x| \leq e^{-1} \).

Interestingly, the function \( f(c) = -2c \cdot e^{-2c} \) gets its minimum at \( c = 0.5 \), where it precisely equals the radius of convergence \( -e^{-1} \). Therefore, for all \( c \) we can substitute \( x = -2c \cdot e^{-2c} \), since we are within the radius of convergence of \( T(x) \), and we finally derive the result. \( \square \)

We note that this particular asymptotic result can be also achieved by the theory of giant components in random graphs [114, 116]. However, this technique is not applicable for finite \( n \) and \( m \), and cannot be used to derive most of the other results in this chapter. We now give an outline of such a proof.

**Proof outline.** Considering the random graph with \( m \) vertices and \( n \) edges such that a vertex \( m_1 \) is connected to vertex \( m_2 \) if and only if there exists an element that hashes into \( m_1 \) and \( m_2 \). This random graph is called the cuckoo graph [37]. Neglecting the \( O(1) \) loops, this graph is equivalent to the Erdős-Renyi random graph \( G_{m,n} \) that assigns equal probability to all graphs with exactly \( n \) edges (and \( m \) vertices)

A matching in \( G_{m,n} \) corresponds to directing some of the edges in the random graph such that the in-degree is at most 1. For each connected component \( C \) in \( G_{m,n} \), if \( C \) is a tree we can direct all edges, while in all other cases we can direct as much edges as the number of vertices.

The number of such edges and vertices can be found in [114, 116], yielding the exact same result. \( \square \)
The following simple illustration of the result shows that any multiple-choice hashing scheme with \( d = 2 \) can only reach about 84% of SRAM occupancy when the load is 1.

**Example 12** In case \( c = 1 \), that is \( n = m \), the normalized limit expected maximum matching size is

\[
\psi = 1 + \frac{1}{2} \cdot W(-2 \cdot e^{-2}) + \frac{1}{4} W^2(-2 \cdot e^{-2}) \approx 0.8381.
\]

The following corollary shows that when the load is below \( \frac{1}{2} \), the probability for a right-side vertex to be part of a maximum matching goes to 1. This corollary also follows from the previously known result that there is a perfect matching with high probability in cuckoo hash tables with load \( c \leq \frac{1}{2} \) [35].

**Corollary 28** Let \( d = 2 \) and \( c = \frac{n}{m} \leq \frac{1}{2} \). Then the limit normalized expected maximum matching size is \( \psi = \lim_{n \to \infty} \frac{\mu(G_n)}{n} = 1 \).

**Proof.** In case \( c \leq \frac{1}{2} \), \( W(-2c \cdot e^{-2c}) \) equals \(-2c\), thus, \( \psi = \frac{1}{c} + \frac{1}{2c} \cdot (-2c) + \frac{1}{4c^2} (-2c)^2 = 1 \)

### 7.4 Low Memory Bandwidth: Bipartite Graphs With Low Memory Bandwidth

In this section we are interested in a *low-memory-bandwidth version* of the hash algorithm. We now let each element choose either 1 or 2 buckets instead of only 2 buckets, to force them to access less buckets and use less memory I/O bandwidth.

The idea behind this algorithm is that it may use less SRAM accesses than a full hashing algorithm. On the other hand, it will be less memory-efficient and therefore will also need to access the DRAM more often. We are interested in the tradeoff between these two considerations.

Formally, we relax the constraint that each vertex in \( L \) chooses exactly 2 vertices in \( R \), and let each left-side vertex choose either 1 or 2 right-side vertices. Since we can divide the set of vertices either deterministically or randomly, we will discuss the results in both cases. See also [117] for a similar model.

#### 7.4.1 Model

**Definition 12** Let \( d_v \) be the number of choices of each vertex \( v \in L \). The average number of choices \( a \) is the average left-side vertex degree, i.e. \( a = \frac{\mathbb{E}(\sum_{v \in L} d_v)}{n} = \frac{\sum_{v \in L} \mathbb{E}(d_v)}{n} \).

First, in the deterministic case, we find the expected maximum matching size of the graph \( G_a = (L + R, E) \), where each vertex \( v \in L \) independently chooses a predetermined number \( d_v \in \{1, 2\} \) of random vertices in \( R \), such that \( a = \frac{d_1 + 2d_2}{n} \).
Second, in the random case, we analyze the slightly different case of a random bipartite graph $G_p = \langle L + R, E \rangle$ where each vertex chooses two vertices with probability $p$ and one vertex with probability $1 - p$. This implies that in $G_p$, the average number of choices $a = 1 + p$.

### 7.4.2 Connected Components in Deterministic Graphs

As in Section 7.3.1, we now consider a deterministic bipartite graph $H = \langle L_H + R_H, E_H \rangle$, with $|L_H| = s$ and $|R_H| = q$. We assume that the degree of each vertex in $L_H$ is at most 2.

**Proposition 1** Lemmas 2, 3, and 4 hold also when the degree of each vertex in $L_H$ is at most (but not necessarily) 2.

Note that the proofs remain almost identical to the original proofs, replacing a few equalities with the corresponding inequalities.

**Lemma 7** Let $s + 1 = q$. If $H$ is connected then the degree of each vertex in $L_H$ is 2.

**Proof.** Assume on the contrary that $H$ is connected but that there is (at least) a single vertex $v_L \in L_H$ with degree 1. Consider the bipartite graph $\hat{H} = \langle \hat{L}_H + \hat{R}_H, \hat{E}_H \rangle$, that is given by removing the vertex $v_L$ (and its connected edge) from $H$. By the construction of $\hat{H}$, we get that $\hat{H}$ is connected, but $|\hat{L}_H| + 1 < |\hat{R}_H|$, which contradicts Lemma 2.  

### 7.4.3 Expected Maximum Matching Size

**Predetermined Number of Choices**—We assume that each vertex $v \in L$ independently chooses $1 \leq d_v \leq 2$ random vertices in $R$, where $d_v$ is predetermined. The following result provides the expected maximum matching size in this case.

**Theorem 29** Given a predetermined average number of choices $a$, let $d_1 = (2 - a) \cdot n$ and $d_2 = n - d_1 = (a - 1) \cdot n$ be the number of vertices in $L$ that choose one and two vertices in $R$, respectively. The expected maximum matching size $\mu(G_a)$ is given by:

$$\mu(G_a) = m - \sum_{s=0}^{\ell} \left\{ \begin{array}{c} \frac{d_2}{s} \cdot \binom{m}{s+1} \cdot \left(1 - \frac{s+1}{m}\right)^{2(d_2-s)+d_1} \\
\binom{s+1}{m}^{2s} \cdot \frac{2s!}{(s+1)^{s+1}} \end{array} \right\}$$

where $\ell = \min (d_2, m - 1)$.

**Proof.** As in the proof of Theorem 24, our proof is based on counting the expected number of vertices in $L$ that are not in some specific maximum matching $M$ of $G$, based on the decomposition of $G$ into its connected components. The proof is almost identical, with the
modification that, due to Lemma 7, we only take into account the \(d_2\) vertices that have a degree of 2 (instead of all \(n\) vertices in the proof of Theorem 24).

Thus, the expected number of connected components in \(G\) with \(s\) elements in \(L\) and \(s + 1\) in \(R\) is given by:

\[
\left( \frac{d_2}{s} \right) \left( \frac{m}{s + 1} \right) \cdot \left( 1 - \frac{s + 1}{m} \right)^{2(d_2 - s) + d_1} \cdot \left( \frac{s + 1}{m} \right)^{2s} \cdot P_s,
\]

where the above expression consists of the same considerations as in the proof of Theorem 24. Finally, as before, adding the expressions for all possible \(s\)'s and subtracting the sum from \(m\) yields the claimed result. \(\square\)

**Random Number of Choices**—We assume that each vertex \(v \in L\) independently chooses \(1 \leq d_v \leq 2\) random vertices in \(R\), where for each \(v \in L\), \(d_v\) equals 2 with probability \(p\), and it equals 1 with probability \(1 - p\). The following result reflects the expected maximum matching size in this case.

**Theorem 30** The expected maximum matching size \(\mu(G_p)\) is given by

\[
\mu(G_p) = \sum_{d_2=0}^{n} \binom{n}{d_2} \cdot p^{d_2} \cdot (1 - p)^{n-d_2} \cdot \mu(G_{a=1+ \frac{d_2}{n}}),
\]

where \(\mu(G_a)\) is given by Theorem 29.

**Proof.** The number of vertices in \(L\) with degree 2 follows a Binomial distribution with \(n\) experiments and a probability of success \(p\). In Theorem 29 we found the expected maximum matching size of each such instance. Thus, by the law of total expectation, the claimed result is given by computing the weighted average, where we compute \(a\) by the equations \(d_1 + d_2 = n\) and \(d_1 + 2 \cdot d_2 = a \cdot n\). \(\square\)

**7.4.4 Limit Normalized Expected Maximum Matching Size**

**Predetermined Number of Choices**—We are also interested in the asymptotic expression, where \(n \to \infty\), such that we fix both the load \(c = \frac{n}{m}\) and the average number of choices \(a = \frac{d_1 + 2d_2}{n}\) of the vertices. This is reflected in the following theorem.

**Theorem 31** The limit normalized expected maximum matching size \(\psi_a = \lim_{n \to \infty} \frac{\mu(G_a)}{n}\) with average number of choices \(a \in (1, 2]\) is given by:

\[
\psi_a = \frac{1}{c} + \frac{W((-2c(a-1) \cdot e^{-ac})}{2c^2 \cdot (a-1)} + \frac{W^2(-2c(a-1) \cdot e^{-ac})}{4c^2 \cdot (a-1)}.
\]

For \(a = 1\), it is \(\psi_a = \frac{1}{c} - \frac{1}{c} \cdot e^{-c}\).
Proof. We compute the limit of \( \frac{\mu(G_n)}{n} \) as \( n \to \infty \). We consider the case where \( c = \frac{n}{m} \) and \( a = \frac{d_1 + 2d_2}{n} > 1 \) are fixed. So \( \psi_a = \lim_{n \to \infty} \frac{\mu(G_n)}{n} \), that is,

\[
\psi_a = \lim_{n \to \infty} \frac{1}{n} \left( m - \sum_{s=0}^{\ell} \binom{d_2}{s} \binom{m}{s+1} \left( 1 - \frac{s+1}{m} \right)^{-2(d_2-s)+d_1} \left( \frac{s+1}{m} \right)^{2s} P_s \right)
\]

Given that \( a = \frac{d_1 + 2d_2}{n} \) and \( n = d_1 + d_2 \), we find that \( d_2 = (a-1) \cdot n \) and \( d_1 = (2-a) \cdot n \). Similarly to the proof of Theorem 27, we first have to find that each term in the summation is an increasing function with respect to \( n \). We discover that \( (1 - \frac{s+1}{m})^{-2(d_2-s)+d_1} = (1 - \frac{s+1}{m})^{a-n-s} \) is an increasing function (using differentiation), and also find that \( \frac{1}{n \cdot (a-1-n) \cdot (m_s^1) \cdot \left( \frac{s+1}{m} \right)^{2s}} \) is an increasing function as previously. Consequently, each term in the sum is an increasing function and, by the monotone convergence theorem [115], we can put the limit inside the sum. By further simplifying the above expression as in the proof of Theorem 27 we eventually get:

\[
\psi_a = \frac{1}{c} - \frac{1}{2c^2 \cdot (a-1) \cdot e^{-ac}} \sum_{j=1}^{\infty} (-j)^{j-2} j! \cdot (-c \cdot 2 \cdot (a-1) \cdot e^{-ac})^j
\]

Let \( T(x) = \sum_{j=1}^{\infty} \frac{(-j)^{j-2}}{j!} \cdot x^j \) be a Taylor expansion, where by substituting \( x = -c \cdot 2 \cdot (a-1) \cdot e^{-ac} \) we get the above expression. Similarly to the proof of Theorem 27, we get that

\[
T(x) = -W(x) - \frac{1}{2} W^2(x),
\]

with convergence within \( |x| \leq e^{-1} \) [97].

Since the function \( f(c) = -c \cdot 2 \cdot (a-1) \cdot e^{-ac} \) gets its minimum at \( c = a^{-1} \), where it equals \( - \frac{2(a-1)}{a} e^{-1} \), and \( - \frac{2(a-1)}{a} e^{-1} \leq e^{-1} \) for all \( a \in [1, 2] \), then for all \( c \) we can substitute \( x = -c \cdot 2 \cdot (a-1) \cdot e^{-ac} \). Hence, it is within the radius of convergence of \( T(x) \).

Finally, for the case where \( a = 1 \), then \( d_2 = 0 \) and \( d_1 = n \). Therefore, the expression for the expected maximum matching size is reduced to \( m - \left( m \cdot \left( 1 - \frac{1}{m} \right)^n \right) \). Thus,

\[
\psi_a = \lim_{n \to \infty} \frac{\mu(G_n)}{n} = \lim_{n \to \infty} \frac{1}{n} \left( m - \left( m \cdot \left( 1 - \frac{1}{m} \right)^n \right) \right)
\]

\[
= \frac{1}{c} - \frac{1}{c} \cdot e^{-c}.
\]

Interestingly, if even a small fraction of the elements do not have choice, then the limit normalized expected maximum matching size is not 1. This is reflected in the following corollary.
Corollary 32 ((No) Perfect Matching) If \(1 \leq a < 2\) then \(\psi_a < 1\).

**Proof.** We show that \(\psi_a\) is strictly monotonically increasing, thus \(\psi_a < 1\) for \(1 \leq a < 2\), since \(\psi_a = 1\) for \(a = 2\). This is shown by differentiating \(\psi_a\) with respect to \(a\):

\[
\frac{d\psi_a}{da} = -\frac{1}{4a^2(a-1)^2} \cdot (W(-2c(a-1) \cdot e^{-ac}) + 2c(a-1)) \\
W(-2c(a-1) \cdot e^{-ac})
\]

Both the first factor \(-\frac{1}{4a^2(a-1)^2}\) and the third factor \(W(-2c(a-1) \cdot e^{-ac})\) are negative. Thus, if the second factor is positive then \(\frac{d\psi_a}{da}\) is an increasing function with respect to \(a \in [1,2)\).

If \(c > 0.5\), then \(2c(a-1) > 1\), and since \(W(x)\) is minimized for \(x = -\frac{1}{e}\) where it equals \(-1\), the second factor is positive. On the other hand, consider that \(c \leq 0.5\). Since \(W(-2c(a-1) \cdot e^{-2c(a-1)}) = -2(a-1)c\) and \(W(x)\) is an increasing function, then we have to show that \(-2c(a-1) \cdot e^{-2c(a-1)} < -2c(a-1) \cdot e^{-ac}\), that is, \(-2c(a-1) > -ac\). The last inequality can easily be shown for \(1 \leq a < 2\). \(\square\)

Random Number of Choices—We now study the case of the random bipartite graph \(G_p = (L + R, E)\), where each vertex chooses two vertices with probability \(p\), and a single vertex with probability \(1 - p\). As we show in the next theorem, the asymptotic expression can be derived from \(\psi_a\).

**Theorem 33** The limit expected maximum matching size \(\psi_p = \lim_{n \to \infty} \frac{\mu(G_p)}{n}\) is \(\psi_p = \psi_{a=1+p}\).

**Proof.** We compute the limit of \(\frac{\mu(G_p)}{n}\) as \(n \to \infty\).

\[
\psi_p = \lim_{n \to \infty} \frac{\mu(G_p)}{n} \\
= \lim_{n \to \infty} \frac{1}{n} \sum_{d_2=0}^{n} \binom{n}{d_2} \cdot p^{d_2} \cdot (1-p)^{n-d_2} \cdot \mu \left( G_{a=1+d_2} \right)
\]

Let \(X \sim \text{Bin}(n, p)\) be the random variable counting the number of vertices in \(L\) that choose 2 vertices in \(R\). By summing over three disjoint ranges of possible values for \(d_2\), we get

\[
\psi_p = \lim_{n \to \infty} \sum_{d_2=0}^{\lfloor np - n^2 \rfloor} \Pr \{X = d_2\} \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+d_2} \right) + \lim_{n \to \infty} \sum_{d_2=\lfloor np + n^2 \rfloor - 1}^{\lfloor np + n^2 \rfloor - 1} \Pr \{X = d_2\} \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+d_2} \right) + \lim_{n \to \infty} \sum_{d_2=\lfloor np - n^2 \rfloor + 1}^{n} \Pr \{X = d_2\} \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+d_2} \right)
\]

117
By Chebyshev’s inequality we get that \( \Pr \left\{ \left| X - np \right| > n^{\frac{1}{4}} \sqrt{np(1-p)} \right\} \leq \frac{1}{n^{1/4}} \). Since \( p(1-p) \leq 1 \), we get that \( \Pr \left\{ \left| X - np \right| > n^{\frac{1}{4}} \right\} \leq \frac{1}{n^{1/4}} \). By the fact that \( \frac{1}{n} \cdot \mu \left( G_{a=1+\frac{d_2}{n}} \right) \leq 1 \), we find that the first and the third limits go to zero.

Since the function \( \mu \left( G_{a} \right) \) is increasing with respect to \( a \) (this can be shown by a simple combinatorial argument), we get the following lower bound:

\[
\psi_p = \lim_{n \to \infty} \sum_{d_2=\lfloor np-n^{3/4} \rfloor+1}^{\lfloor np+n^{3/4} \rfloor-1} \Pr \{ X = d_2 \} \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+\frac{d_2}{n}} \right) 
\geq \lim_{n \to \infty} \left( 1 - \frac{1}{n^{1/4}} \right) \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+\frac{\lfloor np-n^{3/4} \rfloor+1}{n}} \right) 
\]

as well as the following upper bound:

\[
\psi_p = \lim_{n \to \infty} \sum_{d_2=\lfloor np-n^{3/4} \rfloor+1}^{\lfloor np+n^{3/4} \rfloor-1} \Pr \{ X = d_2 \} \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+\frac{d_2}{n}} \right) 
\leq \lim_{n \to \infty} 1 \cdot \frac{1}{n} \cdot \mu \left( G_{a=1+\frac{\lfloor np+n^{3/4} \rfloor-1}{n}} \right) .
\]

By the squeeze theorem, we get the claimed result. \( \square \)

### 7.5 Static Partitioning of 2 Choices

We now consider a popular multiple-choice hashing implementation variant in which the buckets are statically partitioned into two equal sets, and each element holds one hash function to each set. This variant is easier to implement in hardware, because it can be implemented using two simple single-ported memories, instead of a single dual-ported one.

Formally, we consider the random bipartite graph \( G_{\alpha_1} = (L + (R_u \cup R_d), E) \), where \( R \) is now partitioned into two disjoint subsets \( R_u \) and \( R_d \) with \( |R_u| = \alpha_1 \cdot m \) and \( |R_d| = (1 - \alpha_1) \cdot m \). Each vertex \( v \in L \) independently chooses a single random vertex in \( R_u \) and another single random vertex in \( R_d \). This corresponds, for example, to a hashing scheme that selects non-overlapping sets of buckets as images of its hash functions (e.g., as in multilevel hashing scheme [32] or \( d \)-left [26]).

Note that further evaluation of the results reported in this section can be found in Section 7.7.3.

#### 7.5.1 Connected Components in Deterministic Graphs

The following lemma counts all the possible bipartite graphs \( H_{ad} \) of the form \( \langle L_H + (R_{H_u} \cup R_{H_d}), E_H \rangle \) with degree 2 for each vertex in \( L_H \), where \( |L_H| = s \), \( |R_{H_u}| = i \) and \( |R_{H_d}| = j \), such that each vertex \( v \in L_H \) is connected using a single edge to some
vertex in $R_{H_u}$ and another single edge to some vertex in $R_{H_d}$.

**Proposition 2** Let $s = i + j - 1$. The number $T_{i,j}$ of connected bipartite graphs is $T_{i,j} = i^{j-1} \cdot j^{i-1} \cdot s! = i^{j-1} \cdot j^{i-1} \cdot (i + j - 1)!$

**Proof.** The proof is identical to the proof of Lemma 5 with two modifications. First, instead of initially counting the number of trees over the set $R_H$, we count the number of parity trees [118] over the disjoint sets $R_{H_u}$ and $R_{H_d}$. By [118] we are given that the number of parity trees is $i^{j-1} \cdot j^{i-1}$. Second, we do not have to color the edges because of the partition.

### 7.5.2 Expected Maximum Matching Size

In the next theorem we find the expected maximum matching size with a static partition of the right-side vertices.

**Theorem 34** Given the static partitioning of the bipartite graph $G_{\alpha_1}$, the expected maximum matching size $\mu(G_{\alpha_1})$ is

$$\mu(G_{\alpha_1}) = m - \sum_{s=0}^{n} \binom{n}{s} \sum_{i=1}^{\ell_1} \binom{\alpha_1 \cdot m}{i} \binom{(1 - \alpha_1) \cdot m}{s+1-i} \left(1 - \frac{i}{\alpha_1 \cdot m}\right)^{n-s} \cdot \left(1 - \frac{s+1-i}{(1 - \alpha_1) \cdot m}\right)^{s},$$

where $\ell_1 = \max\{0, s+1 - (1 - \alpha_1) \cdot m\}$, $\ell_2 = \min\{s+1, \alpha_1 \cdot m\}$, $P_{i,s+1-i} = \frac{T_{i,j}}{(i+j) \cdot T_{i,j-1}}$, and $T_{i,j} = i^{j-1} \cdot j^{i-1} \cdot (i + j - 1)!$.

**Proof.** Similarly to the proof of Theorem 24, our proof is based on counting the expected number of vertices in $L$ that are not in some specific maximum matching $M$ of $G_{\alpha_1}$, based on the decomposition of $G$ into its connected components. As in the proof of Theorem 24, we consider the number of connected components with exactly $s$ vertices in $L$ and $q = s + 1$ vertices in $R_u \cup R_d$, where we have to sum over all possible combinations $(i, s+1-i)$, where $i$ corresponds to the number of vertices taken from $R_u$ and $s+1-i$ corresponds to those taken from $R_d$.

Thus, the expected number of connected components in $G_{\alpha_1}$ with $s$ vertices in $L$, $i$ vertices in $R_u$ and $s+1-i$ vertices in $R_d$ is given by:

$$\binom{n}{s} \binom{\alpha_1 \cdot m}{i} \binom{(1 - \alpha_1) \cdot m}{s+1-i} \left(1 - \frac{i}{\alpha_1 \cdot m}\right)^{n-s} \cdot \left(1 - \frac{s+1-i}{(1 - \alpha_1) \cdot m}\right)^{s} \cdot P_{i,s+1-i},$$

119
The above expression consists of the following factors (in order):
(i) choosing the \( s \) vertices in \( L \);
(ii) choosing the \( i \) vertices in \( R_u \);
(iii) choosing the \( s + 1 - i \) vertices in \( R_d \);
(iv) the probability that all \( i \) vertices in \( R_u \) may be connected only to the chosen \( s \) vertices in \( L \);
(v) the probability that all \( s + 1 - i \) vertices in \( R_d \) may be connected only to the chosen \( s \) vertices in \( L \);
(vi) the probability that all \( s \) vertices in \( L \) are only connected to the \( i \) vertices in \( R_u \);
(vii) the probability that all \( s \) vertices in \( L \) are only connected to the \( s + 1 - i \) vertices in \( R_d \); and,
(viii) the probability that all chosen vertices are connected.

Finally, adding the expressions for all possible \( s \)'s and \( i \)'s and subtracting it from \( m \) yields the claimed result. \( \square \)

Up until now we were only interested in the maximum matching size. However, to determine the latency and throughput of our hash table, we also need to know how many elements are in each of the two partitions. Note that there are many matchings with the maximum matching size. Among those, we are interested in the matchings that maximize the expected number of elements in the first partition. This is reflected in the following theorem.

**Theorem 35** Given the static partitioning of the bipartite graph \( G_{\alpha_1} \), there is a maximum matching such that the expected number of elements in the first partition is

\[
\mu^1(G_{\alpha_1}) = \alpha_1 m \cdot \left(1 - \left(1 - \frac{1}{\alpha_1 m}\right)^m\right).
\]

Moreover, there is no other maximum matching with a higher expected number of elements in the first partition.

**Proof.** The proof follows the fact that there is a matching whose size is maximum and all buckets in the first partition with at least one element hashed to them are occupied. It follows by considering the connected components in the corresponding bipartite graph. From Proposition 2 it follows that we should care only for the connected components with \( s \) vertices in \( L_H \) and \( s + 1 \) vertices in \( (R_{H_u} \cup R_{H_d}) \). In all other components the number of elements is bigger or equal to the number of buckets and so all buckets are occupied (Proposition 2).

Consider a connected components with \( s \) vertices in \( L_H \) and \( s + 1 \) vertices in \( (R_{H_u} \cup R_{H_d}) \). Further assume a maximum matching (of size \( s \), by Proposition 2), with one vertex \( v_r \in R_{H_u} \) that is not matched. Since the connected component is a tree, there is a path from \( v_r \) to to some other matched vertex in \( R_{H_d} \). Moreover, this path alternates between edges in the matching and edges that are not in the matching. By switching between the two sets of
edges we get a new matching whose size is maximum and all vertices in \( R_{H_u} \) are matched. Since the first partition size is \( \alpha_1 m \), and there are \( n \) elements, the probability that no element hashes into some bucket in the first partition is \( \left( 1 - \frac{1}{\alpha_1 m} \right)^n \). It then follows that the expected number of occupied buckets in the first partition is as claimed in the theorem.

\[ \square \]

### 7.5.3 Limit Normalized Expected Maximum Matching Size

As in the previous sections, we are also interested in the asymptotic best behavior of the partitioned hashing scheme where \( n \to \infty \) with both fixed load \( c = \frac{n}{m} \) and fixed partition \( \alpha_1 \). We obtain the following theorem.

**Theorem 36** Given the static partitioning of the bipartite graph \( G_{\alpha_1} \), the limit normalized expected maximum matching size \( \psi_{\alpha_1} = \lim_{n \to \infty} \frac{\mu(G_{\alpha_1})}{n} \) for \( \alpha_1 \in (0, 1) \) is given by:

\[
\psi_{\alpha_1} = \frac{1}{c} - \frac{\alpha_1 \cdot (1 - \alpha_1)}{c^2} \cdot (t_1 + t_2 - t_1 \cdot t_2),
\]

where \( t_1, t_2 \) are provided by the following equations:

\[
\frac{e}{1 - \alpha_1} \cdot e^{-\frac{\alpha_1}{e}} = t_1 \cdot e^{-t_2}, \quad \frac{c}{\alpha_1} \cdot e^{-\frac{\alpha_1}{e\alpha_1}} = t_2 \cdot e^{-t_1},
\]

and satisfy the condition \( t_1 \cdot t_2 \leq 1 \).

For \( \alpha_1 \in \{0, 1\} \) (namely, the trivial partitions), the limit normalized expected maximum matching size \( \psi_{\alpha_1} \) is \( \frac{1}{c} - \frac{1}{c} \cdot e^{-c} \).

**Proof.** As in the proof of Theorem 27, we compute the limit of \( \frac{\mu(G)}{n} \) as \( n \to \infty \). We consider the case where \( c = \frac{n}{m} \) and \( 0 \leq \alpha_1 \leq 1 \) are fixed. So \( \psi_{\alpha_1} = \lim_{n \to \infty} \frac{\mu(G_{\alpha_1})}{n} \), that is,

\[
\psi_{\alpha_1} = \lim_{n \to \infty} \frac{1}{n} \cdot \left( m - \sum_{s=0}^{n} \binom{n}{s} \cdot \sum_{i=0}^{b_2} \frac{\alpha_1 \cdot m}{i} \cdot \binom{1 - \alpha_1 \cdot m}{s+1-i} \cdot \left( 1 - \frac{i}{\alpha_1 \cdot m} \right)^{n-s} \cdot \left( \frac{s+1-i}{(1-\alpha_1) \cdot m} \right)^{n-s} \cdot P_i, s+1-i \right).
\]

By substituting the expression for \( P_i, s+1-i \) from Theorem 34, and moving \( \binom{n}{s} \) inside the second summation, we get:

\[
\psi_{\alpha_1} = \lim_{n \to \infty} \left( \frac{1}{c} - \frac{1}{n} \cdot \sum_{s=0}^{n} \sum_{i=0}^{s+1} \binom{n}{s} \cdot \left( \frac{\alpha_1 m}{i} \right) \cdot \binom{1 - \alpha_1 \cdot m}{s+1-i} \cdot \left( 1 - \frac{i}{\alpha_1 \cdot m} \right)^{n-s} \cdot \left( \frac{s+1-i}{(1-\alpha_1) \cdot m} \right)^{n-s} \cdot \left( \frac{s+1-i}{i \cdot (s+1-i)} \right)^{i+1} \cdot \left( \frac{s+1-i}{(s+1-i)} \right)^{i+1} \cdot \left( (s+1-i) \cdot (s+1-i-1)! \right) \right).
\]
By substituting $c = \frac{n}{m}$, we get:

$$
\psi_{\alpha_1} = \lim_{n \to \infty} \left( \frac{1}{c} - \frac{1}{n} \sum_{s=0}^{n} \sum_{i=0}^{s+1} \left( \frac{\alpha_1 n}{i} \right) \left( \frac{n-i}{s+1-i} \right) \left( 1 - \frac{i}{s+1} \right)^{n-s} \cdot \left( 1 - \frac{s+1-i}{1+\alpha_1} \right)^{s+1-i} \cdot \left( \frac{i}{s+1} \right)^s \cdot \left( \frac{n-i}{1+\alpha_1} \right)^{s} \cdot \left( \frac{i}{s+1} \right)^{s} \right).
$$

As in the proof of Theorems 27 and 31, using the monotone convergence theorem [115], we can put the limit inside the sum. By further simplifying the above expression with similar consideration to the proofs of Theorems 27 and 31, we get eventually:

$$
\psi_{\alpha_1} = \frac{1}{c} - \frac{\alpha_1 \cdot (1 - \alpha_1)}{c^2} \sum_{s=0}^{\infty} \sum_{i=0}^{s+1} \frac{i^{s+1-i-1} \cdot (s+1-i)^{i-1}}{i! \cdot (s+1)!} \cdot \left( \frac{c}{\alpha_1} \cdot e^{\frac{c}{\alpha_1}} \right)^{s+1-i} \cdot \left( \frac{c}{1 - \alpha_1} \cdot e^{-\frac{c}{1 - \alpha_1}} \right)^i.
$$

We switch the order of summation and get that $i \in \{0, 1, \ldots \}$ and $s$ goes from max\{0, i − 1\} to $\infty$. We also substitute $j = s + 1 - i$ (or $s = i + j - 1$). Thus,

$$
\psi_{\alpha_1} = \frac{1}{c} - \frac{\alpha_1 \cdot (1 - \alpha_1)}{c^2} \sum_{i=0}^{\infty} \sum_{j=\text{max}\{0, i - 1\}}^{\infty} \frac{i^{j-1} \cdot j^{i-1}}{i! \cdot j!} \cdot \left( \frac{c}{1 - \alpha_1} \cdot e^{\frac{c}{1 - \alpha_1}} \right)^j \cdot \left( \frac{c}{\alpha_1} \cdot e^{-\frac{c}{\alpha_1}} \right)^i.
$$

Let $T(x, y) = \sum_{j+i \geq 1} \frac{i^{j-1} \cdot j^{i-1}}{i! \cdot j!} \cdot x^i \cdot y^j$. This expression has been previously found [35] to be the multivariate formal power series about the point $(x, y) = (0, 0)$ of $t(x, y) = t_1(x, y) + t_2(x, y) - t_1(x, y) \cdot t_2(x, y)$ where $t_1(x, y)$ and $t_2(x, y)$ are given by the following implicit multivariate functions:

$$
x = t_1(x, y) \cdot e^{-t_2(x, y)} \quad , \quad y = t_2(x, y) \cdot e^{-t_1(x, y)}
$$

However, the mentioned range of convergence in [35] is insufficient for our case. (Note also that in [35] the sums should be over $i + j \geq 1$ and not over $i, j \geq 0$.)

Since we compute the limit normalized expected maximum matching, then the expression for $\psi_{\alpha_1}$ in Equation (7.3) is bounded from below by 0, thus, by Equation (7.3) the double summation is bounded from above by a constant. On the other hand, all terms in the summation in Equation (7.3) are positive. Then, if we look at the partial-sum series (by defining an arbitrary order), we get an increasing series which is bounded. Thus, by the monotone convergence theorem the double series converges for any values $x$ and $y$ satisfying $x = \frac{c}{1 - \alpha_1} \cdot e^{-\frac{c}{\alpha_1}}$ and $y = \frac{c}{\alpha_1} \cdot e^{-\frac{c}{1 - \alpha_1}}$. 122
However, the multivariate functions in Equation (7.4) have multiple branches (as the Lambert-W function does [97]), that is, for a given \( x \) and \( y \) there is more than one solution. We aim to find this branch in terms of \( t_1 \) and \( t_2 \). We use the implicit function theorem to find the derivatives singularities. The Jacobian is given by

\[
J = \begin{pmatrix}
    e^{-t_2(x,y)} & -t_1(x,y) \cdot e^{-t_2(x,y)} \\
    -t_2(x,y) \cdot e^{-t_1(x,y)} & e^{-t_1(x,y)}
\end{pmatrix},
\]

and it is invertible wherever \(|J| \neq 0\). Thus, there is a derivative singularity in case \( t_1(x,y) \cdot t_2(x,y) = 1 \), which is the only solution. Therefore, as the given formal power series in Equation (7.3) is about the point \((x_0, y_0) = (0,0)\) (which corresponds to \( c = 0 \)), where \( t_1 = t_2 = 0 \), it converges to the branch where \( t_1(x,y) \cdot t_2(x,y) \leq 1 \) (note that both \( t_1(x,y) \) and \( t_2(x,y) \) are always positive).

We deduce the following two corollaries. The first one states that the best performance of multiple-choice hashing scheme with equal partition is asymptotically equivalent to this of a one with no partition. The second one states how close partition needs to be to equal in order to reach an ideal average matching.

**Corollary 37 (Asymptotic Equivalence)** Let \( d = 2 \). The limit normalized expected maximum matching size of \( G_{\alpha_1} \) with \( \alpha_1 = 0.5 \) is the same as the limit expected maximum matching size of \( G \).

**Proof.** We substitute \( \alpha_1 = 0.5 \) in the expression from Theorem 36, and get \( e^{\frac{c}{1-\alpha_1}} \cdot e^{-\frac{c}{\sqrt{1-2\alpha_1}}} = t_1 \cdot e^{-t_2} \cdot \frac{-\alpha_1}{1-\alpha_1} \cdot e^{-t_2} = t_2 \cdot e^{-t_1} \cdot 2 \cdot e^{-t_2}. \) One of the solutions of the above equations is \( t_1 = t_2 = -W(-2ce^{-2c}) \). In the proof of Theorem 27, we showed that \(-W(-2ce^{-2c}) \leq 1 \). Thus, \( t_1 \cdot t_2 < 1 \). By substituting this solution in the expression for \( \psi_{\alpha_1} \) from Theorem 36, we get the exact expression as in Equation (7.1). \( \square \)

**Corollary 38** Let \( d = 2, c \leq \frac{1}{2} \), and fix a partition \( \alpha_1 \). The limit normalized expected maximum matching size \( \psi_{\alpha_1} = \lim_{n \to \infty} \frac{\mu(G_{\alpha_1})}{n} \) is 1 whenever \( 1 - \sqrt{\frac{4c}{1-2c}} \leq \alpha_1 \leq 1 + \sqrt{\frac{4c}{1-2c}} \).

**Proof.** One of the solutions to Equation (7.2) is given by: \( t_1 = \frac{c}{1-\alpha_1} \), \( t_2 = \frac{c}{\alpha_1} \). By substituting \( t_1 \) and \( t_2 \) in the expression for \( \psi_{\alpha_1} \) from Theorem 36, we get that the limit normalized expected maximum matching size is 1. We also have to verify that \( t_1 \cdot t_2 \leq 1 \). Since \( \frac{c}{1-\alpha_1} \) and \( \frac{c}{\alpha_1} \) are both positive, we are left with \( \frac{c}{1-\alpha_1} \cdot \frac{c}{\alpha_1} \leq 1 \). By solving the quadratic inequality, we get the claimed condition. Note that for \( c = 1/2 \) the range reduces to \( \alpha_1 = 1/2 \). \( \square \)

As in the last section, we are also interested in the limit normalized expected fraction of elements in each of the partitions. This following theorem corresponds to Theorem 35.
Theorem 39 Given the static partitioning of the bipartite graph $G_{\alpha_1}$, in the scaled system, there is a maximum matching such that the asymptotic expected fraction of elements in the first subtable is

$$\mu_{\alpha_1}^1 = \frac{\alpha_1}{c} - \frac{\alpha_1}{c} e^{-\frac{\alpha_1}{c}}$$

Moreover, there is no maximum matching with a higher expected fraction.

Proof. The proof is obtained by taking the limit of the expression in Theorem 35, normalized by $n$. \qed

Finally, given the off-chip memory access latency $b$, the following corollary shows the throughput of the hash table. It follows immediately from Theorems 36 and 39,

Corollary 40 Given an on-chip memory with two partitions of sizes $\alpha_1 m$ and $(1 - \alpha_1) m$, and assuming an SRAM of access latency 1 and an off-chip DRAM of access latency $b$, the hash table throughput is

$$
(\mu_{\alpha_1}^1 + 2 \cdot (\psi_{\alpha_1} - \mu_{\alpha_1}^1) + (2 + b) \cdot \psi_{\alpha_1})^{-1}.
$$

(7.5)

Following Corollary 40, it is possible to compute the optimal partition $\alpha_1$ that maximizes the hash table throughput. We further evaluate this in Section 7.7.6.

### 7.6 Bipartite Graphs with More Than 2 Choices

We are now interested in checking how powerful multiple-choice hashing can be when we allow more than 2 hash functions per element. Of course, using more hash functions will result in an increase in implementation complexity, and therefore one goal of this study is to point out the tradeoff between efficiency and complexity.

In this section we briefly show how our method can be applied to find an upper bound on the expected maximum size matching where each left-side vertex has $d > 2$ choices. Formally, we are given two disjoint sets of vertices $L$ and $R$ of size $n$ and $m$, respectively, and a random bipartite graph $G^d = (L + R, E)$, where each vertex $v \in L$ has $d$ outgoing edges whose destinations are chosen independently at random (with repetition) among all vertices in $R$. We obtain the following upper bound on the maximum matching size of the bipartite graph $G^d$.

Theorem 41 Let $\ell = \min \left( n, \left\lceil \frac{m-1}{d+1} \right\rceil \right)$ and $q = (d - 1) \cdot s + 1$. Then, $\mu(G^d)$ is at most

$$
\min \left( n, m - \sum_{s=0}^{\ell} (q-s) \binom{n}{s} \binom{m}{q} (1 - \frac{q}{m})^{d(n-s)} \left( \frac{q}{m} \right)^{ds} \frac{d^s q^l}{q^{d-1}(d-1)^{s+1}} \right).
$$

Proof. We first establish a few lemmas before proving the result. As before, we start by considering a deterministic bipartite graph $H = (L_H + R_H, E_H)$ with degree $d$ of each vertex in $L_H$, where $|L_H| = s$ and $|R_H| = q$. 

124
Lemma 9 \textbf{If} \((d-1) \cdot s \leq q - 2\), \textbf{then} \(H\) \textbf{is not connected}.

\textbf{Proof.} As in the proof of Lemma 2, the proof follows by induction on \(s\). For \(s = 1\), there are \(d\) edges in the graph and therefore every graph with \(q \geq d + 1\) is not connected. Assuming that the claim holds up until \(s = s'\), we next prove that it holds for any bipartite graph \(H'\) such that \(|L_{H'}| = s' + 1\) and \(|R_{H'}| \geq (d-1) \cdot (s' + 1) + 2\). Assume towards a contradiction that there is a graph \(H'\) which is connected.

We first show that there are \(d-1\) vertices \(v_1, v_2, \ldots, v_{d-1}\) in \(R_{H'}\), all of a degree 1 such that they are connected to the same vertex \(v_{\ell} \in R_{H'}\): The sum of right-side vertex degree is \(d \cdot (s' + 1)\). Also, since the graph is connected there are no right-side vertices with degree 0. This implies that there are at least \((d-2) \cdot (s' + 1) + 2\) vertices of degree 1, thus there exists a vertex \(v_{\ell} \in R_{H'}\) as claimed.

By the induction hypothesis, the graph induced by \(L_{H'} \setminus \{v_{\ell}\}\) and \(R_{H'} \setminus \{v_1, v_2, \ldots, v_{d-1}\}\) is not connected, which implies that it has at least two connected components. In \(H'\), \(v_{\ell}\) is connected to all vertices \(v_1, v_2, \ldots, v_{d-1}\). Since its degree is \(d\) it can be connected only to one of these components. This implies that \(H'\) is not connected as well, and the claim follows. \(\square\)

Lemma 10 \textbf{If} \(H\) \textbf{is connected} and \((d-1) \cdot s = q - 1\) \textbf{then} \(\mu(H) = s\).

\textbf{Proof.} Assume towards a contradiction that \(\mu(H) < s\), and consider some maximum matching \(M\). Let \(v_{\ell} \in L_H\) be a vertex that is not in the maximum matching \(M\), and \(v_1, v_2, \ldots, v_{d-1}\) be the vertices in \(R\) (which are not necessarily distinct) that are connected to \(v_{\ell}\). All vertices \(v_1, v_2, \ldots, v_{d-1}\) are connected also to another vertex in \(L_H\), otherwise \(v_{\ell}\) was in the maximum matching \(M\).

Consider the bipartite graph \(\hat{H} = \langle \hat{L}_H + \hat{R}_H, \hat{E}_H \rangle\), which is given by removing \(v_{\ell}\) from \(H\). Since the right-side vertices \(v_1, v_2, \ldots, v_{d-1}\) are also connected to the other left-side vertices (except \(v_{\ell}\)), the bipartite graph \(\hat{H}\) is connected. However, we get that \(|\hat{L}_H| = s - 1\) and \(|\hat{R}_H| = (d-1) \cdot s + 1\), which contradicts with Lemma 9. \(\square\)

We note that in contrast to Lemma 3, the corresponding proposition is not true for \(d > 2\); that is, if \(H\) is connected and \(s \leq q\), then the maximum matching size is not necessarily \(s\). As a counter example, consider the case where \(d = 3\) and \(s = q = 3\), where two left-side vertices choose the same single right-side vertex (using all their 3 choices), and the other left-side vertex chooses all 3 right-side vertices. The resulting bipartite graph is clearly connected, but the maximum matching size is only 2 (only one of the first two left-vertices can be in the matching).

Lemma 11 \textbf{If} \((d-1) \cdot s = q - 1\) \textbf{then} \(H\) \textbf{is connected if and only if it is a tree.}

\textbf{Proof.} The proof consists of the exact same construction \(\hat{H}\) as in the proof of Lemma 5, where we eventually get a contradiction with Lemma 9. \(\square\)
Lemma 12 The number $T^d_s$ of connected bipartite graphs $H$ whose $|L_H| = s$ and $|R_H| = 2(d - 1) \cdot s + 1$ is 

$$T^d_s = \frac{((d-1)s+1)!}{((d-1)!)^s} ((d-1) \cdot s + 1)^{s-2}.$$ 

Proof. By Lemma 11, we have to count the number of bipartite trees over the two disjoint sets $L_H$ and $R_H$ of size $s$ and $(d - 1) \cdot s + 1$. Since $H$ is a tree, then there are no cycles. Consequently, each one of the vertices in $L_H$ is connected to $d$ distinct vertices in $R_H$. Moreover, no two vertices in $L_H$ share more than 1 vertex in $R_H$. For each vertex $v_\ell \in L_H$, let $S_{v_\ell}$ be the set of the $d$ right-side vertices that $v_\ell$ is connected to and also let the cycle $C_{v_\ell}$ be a cycle that consists of the $d$ vertices of $S_{v_\ell}$.

Consider the graph $\tilde{H} = \langle \tilde{R}_H, \tilde{E}_H \rangle$, which is given by connecting each cycle $C_{v_{\ell_1}}$ to $C_{v_{\ell_2}}$ using a common vertex $v_r$ if and only if $v_r$ is connected to both $v_{\ell_1}$ and $v_{\ell_2}$. The resulting graph $\tilde{H}$ is a Husimi graph over $(d-1)s + 1$ vertices, where the number of such (labeled) graphs is $\frac{((d-1)s+1)!}{((d-1)!)^s} ((d-1) \cdot s + 1)^{s-2}$ [119].

Finally, each set $S_{v_\ell}$ is determined by the (labeled) vertex in $R_L$. Thus, we multiply by $s!$ the above expression. \qed

We are now able to prove the result.

Let $M$ be a maximum matching of $G$. Similarly to the proof of Theorem 24, the proof is based on counting the expected number of vertices in $R$ that are not part of $M$, and on the decomposition of $G$ into its connected components.

We count the expected number of connected components with $s$ left-side vertices and $q = (d - 1) \cdot s + 1$ right-side vertices. By Lemma 10, the maximum matching size of each such connected component is exactly $s$. Thus, there are $q - s$ right-side vertices that are not in $M$.

Let $H$ be a bipartite graph $H = \langle L_H + R_H, E_H \rangle$, with degree $d$ for all vertices in $L_H$, where $|L_H| = s$ and $|R_H| = q$. The probability $P_s$ that $H$ is connected is given by $P_s = \frac{(d)!^{T^d_s}}{q^{d-1}}$.

The remainder of the proof is similar to the proof of Theorem 24. \qed

An evaluation of the upper bound and a comparison to the simulated expected matching size is presented in Section 7.7.4.

7.7 Evaluation and Experiments

We now evaluate our theoretical results, using both synthetic evaluations and trace-based experiments.

7.7.1 Expected Maximum Matching Size With $d = 2$

Figure 7.3 shows the expected maximum matching size normalized by $n$ for various values of $n$ and $m$. It compares simulation results with our analytical model from Theorem 24. For each instance of $n$ and $m$, we randomized 10,000 bipartite graphs, then computed the
average value. The results confirm that our model is fairly accurate, and also show the convergence of the expected maximum matching size to its limit.

Figure 7.4 shows the expected maximum matching size normalized by \( n \) as found in Theorem 27, for various values of load \( c \), both via our analytical model and via simulations. The simulations were performed using \( m = 1000 \) and \( n = c \cdot m \). For each value of \( c \), we randomized 100 bipartite graphs. Again, the model appears fairly accurate.

### 7.7.2 Expected Maximum Matching Size With \( d_v \leq 2 \)

Figure 7.5 shows the normalized limit expected maximum matching size, for various values of load \( c \) and average number of choices \( a \), both via our analytical model (from Theorem 31) as well as via simulations. The simulations were performed using \( m = 1000 \) and \( n = c \cdot m \), where for each instance of the simulation we randomized 100 bipartite graphs. Once again, the results confirm that our model is fairly accurate.
7.7.3 Expected Maximum Matching Size With Static Partition

Figure 7.6 shows the limit expected maximum matching size normalized by $n$, for various values of load $c$ and partition $\alpha_1$, both via our analytical model (from Theorem 36) and via simulations. The simulations were performed using $m = 1000$ and $n = c \cdot m$. For each pair of values of $c$ and $\alpha_1$, we randomized 100 bipartite graphs. The results confirm the accuracy of our model. They also illustrate how the limit expected maximum matching size is symmetric around $\alpha_1 = 0.5$, and the symmetric partition case reaches the optimum for this metric.

Note that in case $c = 0.5$ and $\alpha_1 < 0.5$, while it seems that the normalized limit expected maximum matching size is 1, it is not the case. For instance, in case $c = 0.5$ and $\alpha_1 = 0.45$, we get that $1 - \psi_{\alpha_1} \approx 1.675 \cdot 10^{-7}$. Referring to Corollary 38, this is because $\frac{1+\sqrt{1-4c^2}}{2} = \frac{1}{2}$ in that extreme case. For strictly smaller loads, the imbalance in the partition sizes does
7.7.4 Expected Maximum Matching Size With $d > 2$

We evaluate the upper bound found for the expected matching size (Theorem 41). Figure 7.7 shows our upper bound as well as simulation results for various values of the number of choices $d$. We took $n = m = 100$, while for each instance of $d$, we randomized $10^5$ bipartite graphs. In the case of $d = 2$, our upper bound matches the exact expression found in Theorem 24 and thus matches the simulation results. In addition, we can compare simulation results for higher values of $d$ with our bounds. For instance, in the case of $d = 3$ the normalized expected maximum matching size via the simulation is 0.9402, while our upper bound is 0.9508. In case $d = 4$, we get a simulation value of 0.9795, while the corresponding upper bound is 0.9820.

7.7.5 Trace-Driven Experiments

As in the previous chapters, we have also conducted experiments using real-life traces recorded on a single direction of an OC192 backbone link [98], where packets are hashed using a real 64-bit mix function [99]. Our goal is two-folded. First, we would like to verify that our analysis agrees with results of real-life traces. And second, we want to verify that the distribution of the overflow list size is highly concentrated around its mean, as stated in Theorem 25.

We took $m = 10,000$, and set a number of elements $n$ as corresponding to various values of load $c$. We repeated each experiment 100 times. Fig. 7.8 shows that the results of our experiments are very close to our model. Furthermore, it also shows that the minimum and the maximum off-chip DRAM size are close to the mean.
Figure 7.8: Experiment using real-life traces and hash functions of the normalized number of elements in the DRAM, and comparison to the theoretical model.

Figure 7.9: Expected access throughput given that the off-chip memory is $b = 5$ times slower than the on-chip memory.

7.7.6 Evaluation of Access Throughput

We now compare the access throughput of network hash tables using our suggested method whose performance is found in Corollary 40.

Figure 7.9 plots the access throughput when the off-chip memory is $b = 5$ times slower than the on-chip memory. In the case of the partitioned hashing with $d = 2$, it assumes an optimal partitioning for each load, as provided by by Corollary 40. It is clear that there is limited difference between the cases of $d = 2$ with partitioning and $d = 2$ without partitioning, and therefore by simplicity we only consider one of them below. More importantly, it shows that up to a load of approximately 0.7, and for loads above approximately 1, it is better to use $d = 2$ than $d = 3$. Intuitively, this is because the decrease in the number of needed SRAM accesses more than compensates for the resulting loss of efficiency and therefore the increase in the number of DRAM accesses.

Figure 7.10 further highlights this result. It plots the ratio of the throughput in the cases of $d = 3$ and $d = 4$ by the throughput of our suggested method with $d = 2$ and optimal partitioning. It also plots the corresponding ratio when $d$-left [26] is used to utilize the memory (instead of maximum matching as suggested). Therefore, Figure 7.10 emphasizes
the throughput gain (or loss) with respect to our method.

In Figure 7.10(a), when DRAM accesses are $b = 5$ slower than SRAM accesses, we can see more clearly the outperformance of our suggested algorithm with $d = 2$ under most loads, and compared to most hashing algorithms. For instance, at load $c = 1$, i.e. when the SRAM is highly loaded, our algorithm with $d = 2$ is (surprisingly) better then $d = 3$. By Theorem 36 and Theorem 39, we can find that the optimal memory partitioning is $\alpha_1 = 57.0\%$ for the first subtable and $1 - \alpha_1 = 43.0\%$ for the second subtable, and that 47.2\% of the elements are stored in the first subtable, 36.2\% in the second one, and 16.6\% in the off-chip DRAM. The resulting throughput is $(1 \cdot 0.472 + 2 \cdot 0.362 + (2 + 5) \cdot 0.166)^{-1} \approx 0.4241$. This is indeed higher than the throughput with $d = 3$ with no partitioning, which is found to be 0.4194 in simulations.

Finally, Figure 7.10(b) shows that for $b = 15$, the cost of DRAM accesses becomes higher, and therefore there is more incentive to be efficient in the SRAM. Using $d = 3$ outperforms other settings in most cases. Most often, the cost of DRAM is not high enough to justify $d = 4$. 

Figure 7.10: Ratio of the throughput in the cases of using maximum size matching (MSM) with $d = 3$ and $d = 4$ by the throughput of optimally-partitioned maximum size matching based hashing with $d = 2$. A comparison with $d$-left hashing scheme is also plotted.
Chapter 8

Table Distribution Across the Network

In this chapter we present the Palette distribution framework for decomposing large tables into small ones and then distributing them across the network, while preserving the overall functionality. We develop our framework in the context of software-defined networks (SDNs).

In software-defined networks (SDNs), the network controller first formulates abstract network-wide policies, and then implements them in the forwarding tables of network switches. However, fast SDN tables often cannot scale beyond a few hundred entries. This is because they typically include wildcards, and therefore are implemented using either expensive and power-hungry TCAMs, or complex and slow data structures.

Software-defined networking (SDN) in general, and OpenFlow [120, 121] in particular, provide an abstraction of network devices and operations. This abstraction eases the development of new network protocols and policies. These protocols are implemented through the network controller, a single centralized device with a global view of the entire network. The network controller can be seen as a compiler that translates the abstract policies provided by network designers into specific rules in the table of each network switch.

Previous works typically assumed that the table of each switch can hold an infinite number of rules, which makes the compiler easy to design. In practice, however, this assumption does not hold, and the switch table sizes can become a significant bottleneck to scaling SDN networks. We note that many of these tables are implemented using ternary content-addressable memory (TCAM), which is extremely power-hungry and therefore of limited size. Typical implementations of OpenFlow, for example, limit the number of entries in each such table to only 750 [19], while handling about 100,000 concurrent flows.

Our framework, called Palette, helps balance the sizes of the tables across the network, as well as reduce the total number of entries by sharing resources among different connections. It copes with two NP-hard optimization problems: Decomposing a large SDN table into equivalent subtables, and distributing the subtables such that each connection traverses...
each type of subtable at least once. To implement the Palette distribution framework, we introduce graph-theoretical formulations and algorithms, and show that they achieve close-to-optimal results in practice.

We next introduce the Palette framework for distributing these rules into a network of heterogeneous switches with tables of limited size, while preserving the semantics of the SDN policy. The Palette distribution framework is generic in the sense that it does not rely on the exact meaning of the rules, as long as the rules do not determine the routing of the packet. More specifically, the controller application should only specify whether the policy is routing/forwarding-agnostic or not, and should not deal with the implementation complexity of the distribution across the network switches. This is especially useful when the network topology changes, or equipment is replaced.
8.1 The Palette Framework

We turn now to describing our proposed Palette distribution framework. We define an SDN policy as a collection of rules. Each rule consists of a \((\text{pattern}, \text{action})\) pair, i.e. a pattern of specific bits in the packet header along with an action to take upon a pattern match (e.g. drop the packet or increment a counter of some measurement). For aggregation purposes, \textit{don’t-care} bits, denoted by “*”, are allowed in the pattern. Therefore, a given packet header may match more than one rule, and in that case an action is taken according to the highest-priority rule. Typically, the SDN tables evolve over time (that is, new rules are added and some rules are deleted). In addition, occasionally a switch can send notifications (e.g., measurements taken in one of its built-in counters) to the controller.

Palette takes advantage of the fact that the controller has a global view of the network, and therefore, knows the paths taken by all packets. This allows us to share resources among different paths in an efficient way, by using the same rules for different paths in any common switch.

Fig. 8.1 illustrates our approach in a common setting used for access control. Access control consists of determining whether a given packet is permitted in the network or should be dropped. It is usually made by a switch, a router, or a designated Network Intrusion Detection/Prevention System (NIDS/NIPS) middlebox \textit{at the edge of the network}. Specifically, as illustrated in Fig. 8.1(a), some access control is performed on all ingress nodes of the network. In an SDN setting, this typically translates into installing a table with all access control rules in all ingress nodes. However, as shown in Fig. 8.1(b), using our Palette framework, the rules can be distributed across the network switches.

Note that an orthogonal approach based on cSamp [75] would have been to divide the traffic among switches (that is, each switch deals only with part of the packets), by applying at each switch a hash function on certain packet header fields. Then, each switch handles a different range of hash values. This approach improves the data forwarding, as it reduces the load on each switch. However, it does not solve our problem, since each switch should still store the entire table, as the partitioning of the traffic does not take the rules into account.

8.2 Summary of Our Results

In this chapter, we show how to split the rules across the network, such that each switch will have a smaller SDN table. Our basic approach is to divide this problem into two separate subproblems.

The \textit{first subproblem} is to decompose a large table that contains all rules into a predetermined number of smaller tables. We denote each smaller table by a different color. Our decomposition ensures that \textit{the Palette implementation preserves the overall network behavior}. In particular, it has the following two properties.

- \textit{Order-oblivious}: The order in which the smaller tables are accessed does not change
the global action of the network.

- **Semantically-invariant:** This global action of the network is the same as the one taken when using the initial single large table.

Naturally, this implies that not all actions can be decomposed and distributed: for example, a forwarding or a routing action (e.g., “send the packet through port i”) must be taken in its original switch. We assume that the controller application specifies whether the action is safe to decompose or not. Incidentally, this subproblem is also useful in other contexts, such as achieving parallelism and power-efficiency in TCAMs [14, 15].

After obtaining a set of small tables (or colors), the **second subproblem** is to ensure that each packet traverses all the types of small tables (i.e., all colors), so that the resulting setting would be semantically equivalent to a single lookup in the large table.

We model this second problem in a graph-theoretic manner, in which the switches are nodes and the links are edges. We assume that all the possible end-to-end paths are known to the controller. Thus, our goal is to assign colors to the nodes such that each path is a *rainbow path*—a path whose nodes include all the colors. The objective is to maximize the number of colors that we can use, and therefore, to minimize the table size residing at each node.

Although constructing an optimal assignment is NP-hard, we show that when the network is a tree (which corresponds to datacenters) the problem is tractable. We then propose and evaluate sub-optimal greedy algorithms, and show that they achieve close-to-optimal results in practice.

In addition, we also study the multicolored node case where more than one type of subtable can reside in a single node; we show how this additional degree of freedom enhances the flexibility of our Palette distribution framework.

Finally, we evaluate the performance of our greedy algorithms, both in dividing TCAMs and in distributing tables across the network.

We note that dividing the problem into these two subproblems is done since both problems are fundamental and may be used in other contexts. In addition, we believe that it simplifies the presentation and evaluation of our Palette distribution framework. However, a joint optimization may yield better (yet more complex) results. We leave this joint optimization to future research.

**Chapter Organization** In Section 8.3, we deal with our first subproblem, that is, we show how the large TCAM classifier can be divided into smaller TCAMs. In Section 8.4, we give the formulation of the second subproblem, which is NP-hard, analyze some special cases that can be solved efficiently, and present efficient greedy algorithms for approximating the general case. In Section 8.5, we consider cases where more than one color can be assigned to each of the switches. Finally, Section 8.6 provides experimental results.


8.3 Order-Oblivious Table Decomposition

This section analyzes two approaches to dividing a large table into \( w \) subtables: the \textit{Pivot Bit Decomposition} (PBD) and the \textit{Cut-Based Decomposition} (CBD).

8.3.1 Decomposition Rules

Before starting, note that we only divide rules that correspond to policies which are \textit{marked as safe to divide}. The rest of the rules remain in the corresponding sub-table, and will be re-composed with the rules assigned to that table after decomposition. There is a large body of work of how to compose several policies in one table (e.g., using a Cartesian product of the rules \([122, 123]\)).

Hence, we are left with an arbitrary table that is safe to divide. We assume that this table must be able to match all possible strings. For this, we distinguish between \textit{default} and \textit{non-default} rules. The default rule consists of \textit{don’t-care} bits only, and it uses a default blank action (e.g., a \textit{permit} action in ACLs). The non-default rules are all other rules in the table. Clearly, if a default rule exists in the original table, it may be placed only at the end of the table. To follow our convention, after the decomposition we add a default rule automatically to each of the resulting tables.

We further note that after decomposing the original table, the resulting tables may not be optimal. Therefore, it is possible to apply \textit{logic minimization} on the resulting tables using off-the-shelf solutions developed in the context of TCAMs (e.g. \([124, 125, 126]\)).

The correctness of the decomposition implies that each string that matches a non-default rule in the original table, must match a non-default rule in exactly one of the subtables (and the default rule in the other resulting tables). Moreover, strings that match the default rule in the original table, must also match the default rule in all subtables.

8.3.2 Pivot Bit Decomposition

The first method, called \textit{Pivot Bit Decomposition (PBD)}, works by iteratively decomposing one table into two equivalent tables, thus increasing the total number of tables by 1.

This iterative decomposition is done by selecting one \textit{pivot bit} (equivalently, one column) in the table, and splitting the rules into two sets: the first table holds all rules in which the pivot bit is 0, while the second table holds all rules in which the pivot bit is 1. Rules in which the pivot bit is \textit{“don’t care” (“*”)} are rewritten as two complementary rules: one in which the pivot bit is replaced by 0 (and therefore, is part of the first table) and another in which it is replaced by 1 (and therefore, is part of the second table).

Note that while PBD decomposes the table along certain bits like previously-known methods \([14, 15]\), it does not predetermine some \( P \) pivot bits and then decomposes the table into the corresponding \( 2^P \) tables. On the contrary, it adds a single new subtable at each iteration. Therefore, the different subtables may have resulted from different sets of pivot bits.
Naturally, the efficiency of the decomposition depends on the joint selection of the table and of the pivot bit at each iteration. Our goal is to greedily minimize the maximum table size among all the small tables. Therefore, we always decompose the largest table. The pivot bit is then selected to minimize the size of the largest table among the two resulting subtables.

Specifically, for $b \in \{0, 1, *\}$, let $n_i(b)$ be the number of bits with value $b$ in the $i$-th column (that is, the number of rules whose $i$-th bit is $b$). Then, the pivot bit is:

$$\text{pivot} = \arg\min_i (\max\{n_i(0) + n_i(*), n_i(1) + n_i(*)\})$$

$$= \arg\min_i (n_i(*)) + \max\{n_i(0), n_i(1)\} \quad (8.1)$$

Finally, note that if after the decomposition and possibly the logic minimization, the maximum table size is not reduced, we refrain from decomposing the table and move to the next table. The process ends either when we reach $w$ tables, or when all possible decompositions do not result in table size reduction.

**Example 13** We demonstrate our decomposition using the example table depicted in Fig. 8.2. Assume we want to divide the table into $w = 2$ partitions.

We disregard the default rule ($\varphi_7$), and choose bit number 1 as the pivot bit, since it minimizes the expression in Equation (8.1). Rule $\varphi_2$ has * in bit 1, and therefore it is duplicated to rule $\varphi'_2 = 001***0$ and $\varphi''_2 = 011***0$. After the division, the rules $\varphi_1, \varphi'_2,$ and $\varphi_6$ are assigned to the first sub-table, while the rules $\varphi''_2, \varphi_3, \varphi_4,$ and $\varphi_5$ are assigned to the second sub-table. We also need to add default rule $\varphi_7$ to both of the resulting sub-tables, so their final sizes would be 4 and 5, respectively.

We next show the correctness of our decomposition and the fact that it is order-oblivious.

**Lemma 13** If a non-default rule is returned when applying a packet header $h$ on the original table, then, after the PBD is applied, there is exactly one table that returns a non-default rule for $h$; all the other tables will return the default rule.

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\varphi_1 & * & 0 & 1 & 0 & * & 0 \\
\varphi_2 & 0 & * & 1 & * & * & * & 0 \\
\varphi_3 & * & 1 & * & * & 1 & 0 & 1 \\
\varphi_4 & 1 & 1 & 1 & * & 1 & * & * \\
\varphi_5 & 1 & 1 & * & 0 & * & * & * \\
\varphi_6 & 1 & 0 & 0 & 1 & 0 & 1 & * \\
\varphi_7 & * & * & * & * & * & * & * \\
\end{array}
\]
Proof. Recall that packet headers are binary strings (that is, they do not contain *’s). In addition, notice that our decomposition procedure induces a binary tree structure among the tables, where each node in the tree represents a table, whose two descendants are the tables resulting in the decomposition; the root of the tree is the original table, while the leaves of the tree are the \( w \) subtables.

Suppose that two non-default rules are returned from two different tables \( T_0, T_1 \), and let \( T' \) be the deepest common ancestor of the tables. Let \( i \) be the pivot bit selected in decomposing \( T' \). Thus, all rules (but the default rule) of \( T_0 \) differ from the rules in \( T_1 \) by the pivot bit. Let \( T_b \) be the table with value \( b \) in the \( i \)-th bit of the non-default rule. Let \( b' \) be the value of the \( i \)-th bit of \( h \), therefore \( h \) cannot match any non-default rule in \( T_{1-b'} \) and hence a contradiction.

We next show that at least one table returns a non-default rule for \( h \). Such a table can be found by traversing the decomposition tree according to the value of the pivot bits in \( h \). At each node \( T \), whose pivot is \( i \), we check the \( i \)-th bit in \( h \) and go to either \( T_0 \) or \( T_1 \) according to this value; the process is stopped when reaching a leaf \( T' \). Let \( I \) be the set of pivot indices along the path between \( T' \) and the root of the decomposition tree. Note that by construction, the value of these \( |I| \) bits in \( h \) and in all non-default rules of \( T' \) is the same. Assume that \( h \) matches some rule \( \varphi \) in the original table, let \( I_\ast \) be the set of indices in this rule that are \( \ast \), and let \( I_{01} \) be the set of indices in this rule that are either 0 or 1. Since \( h \) matches this rule, \( h \) was carried on to \( T' \) for all pivots in \( I \cap I_{01} \), and was duplicated (with the same action) for all pivots in \( I \cap I_\ast \). Thus \( h \) matches the rule also in \( T' \).

Note that since this claim is about the semantics of the rules, the claim still holds even when considering the logic minimization at each decomposition step, since logic minimization must preserve the semantics of each table. \( \square \)

The next lemma complements Lemma 13 and shows that correctness is maintained when matching the default rule:

**Lemma 14** If the default rule is returned when applying a packet header \( h \) on the original table, then all tables return the default rule.

**Proof.** It is straightforward since no rule matches the packet header \( h \) in the original table, \( \square \)

The next theorem establishing the correctness of PBD follows immediately from Lemma 13 and Lemma 14:

**Theorem 42** The PBD scheme preserves the semantics of the original table, no matter the order in which the tables are accessed.

We point out two main drawbacks in the basic PBD scheme. First, the basic PBD scheme divides the table at each iteration so that the maximum size of the resulting two
subtable sizes is minimized. Therefore, the sizes of the resulting two subtables after each iteration tend to be almost equal. As a result, when \( w \) is not a power of two, it is expected that the partition sizes would be *imbalanced*.

To solve this problem, we generalize the PBD scheme in the following way: Given the target number of subtables \( w \), we first find the largest integer \( p \) such that \( 2^p < w \). Then, we find a pivot bit that attempts to divide the table such that the ratio between the resulting table sizes will be \( 2^p : (w - 2^p) \).\(^1\) We recursively use this generalization of PBD on each of the two subtables, aiming to decompose the first subtable into \( 2^p \) smaller subtables, and the second subtable into \( w - 2^p \) ones.

Another potential drawback of PBD is reflected in the following result, which exhibits an example in which the largest resulting subtable is of size \( N - w + 1 \), while the optimal decomposition would have resulted in \( w \) tables of size \( N/w \).

**Theorem 43** PBD may result in a decomposition whose largest subtable is asymptotically \( w \) times larger than the largest subtable in the optimal decomposition.

**Proof.** Our counterexample is based on rules that do not contain \( \bullet \) bits at all. Since there is no dependency between the rules (each rule matches exactly one key), any partitioning of the rules into \( w \) sets of size \( N/w \) is a valid decomposition.

On the other hand, consider the set of rules \( \{ \varphi_i = 1^01^W_{-1-i} \mid i \in \{0, \ldots, W - 1\} \} \), where \( W \) is the width of the table. Namely, \( \varphi_i \) is a rule whose all bits are 1 except the \( i \)-th bit, which is set to 0. Assume our PBD chooses the bits \( \{i_1, \ldots, i_{w-1}\} \) during its execution. The resulting subtables will be \( w - 1 \) singleton tables: \( \{\varphi_{i_1}\}, \ldots, \{\varphi_{i_{w-1}}\} \), while the rest of the \( N - w + 1 \) rules will reside in one table. \( \square \)

### 8.3.3 Cut-Based Decomposition

We now offer a second approach to decomposing the table, called *cut-based decomposition* (CBD). This decomposition is based on representing the set of rules in a directed dependency graph.

As illustrated in Fig. 8.3, which shows the dependency graph of the table in Fig. 8.2, the nodes in this graph represent the rules. Moreover, there is an edge from node \( u \) to node \( v \) if and only if rule \( u \) has higher priority than rule \( v \), and there is at least one key that matches both rules. Namely, the edges of the graph represent dependencies between the rules. Our goal is to decompose the graph, which corresponds to the original table, into component subgraphs, which will correspond to the resulting subtables, such that there are no edges between the components. That is, no key matches rules in different components.

First, since all rules match the default rule, and it is allowed to match the default rule in all subtables, we omit the node corresponding to this default rule in the graph.

---

\(^1\)For example, for \( w = 7 \), the goal is to have two tables, one holding approximately \( 4/7 \) of the entries and the other \( 3/7 \). Thus the ratio between the tables is \( 4:3 \), while the basic PBD scheme aims to achieve a ratio \( 1:1 \) between the tables.
Second, we assign a weight to each edge. The weight corresponds to the cost of breaking this edge: an edge can be broken by changing the rules in such a way that no dependency remains between the rules, and the semantic is preserved.

Specifically, let $b^i_u$ denote the $i$-th bit of node $u$. For any node $v$, define the following set of dependency bits: $C_{u,v} = \{ i \mid b^i_v = \ast \text{ and } b^i_u \neq \ast \}$. The weight of the edge between node $u$ and node $v$, denoted by $w(u,v)$, is $|C_{u,v}| - 1$. The weight $w(u,v)$ corresponds to a possible way of resolving the dependency between $u$ and $v$ by adding $w(u,v)$ nodes to the graph: for each bit $i$ in $C_{u,v}$, we can write a rule that is identical to $v$, except the $i$-th bit that is replaced by $1 - b^i_u$. These rules do not have a dependency with $u$. In addition, each key that matches $v$ in the original rule-set will match at least one of these rules. Note that when removing a single edge from the graph, we create a new graph: edges that touch node $v$ in the original graph might be duplicated to the new $|C_{u,v}| - 1$ nodes; the weight of these duplicated edges can only decrease.

Another operation that we also allow in this scheme is a node expansion, that is, given a set of $t \ast$ bits in some rule, we replace the rule with $2^t$ new rules by replacing the $\ast$ bits with a binary enumeration of possible combinations of 0s and 1s. By definition, this operation does not change the semantics of the original table. However, it reduces the connectivity of the dependency graph, facilitating the graph partitioning.

The next theorem shows that CBD is in fact a generalization of PBD:

**Theorem 44** Using the above-mentioned edge breaking and node expansion operations, CBD can exactly emulate PBD.

**Proof.** When PBD partitions a table into two tables using some pivot bit, it duplicates the rules with a *don’t-care* ($\ast$) value in the corresponding bit into the two resulting subtables, and replaces the corresponding $\ast$ value by 0 and 1, respectively. This corresponds to a rule expansion in the CBD. This way, CBD can mimic the exact steps of PBD. Since at
each step in PBD, no packet header matches two rules from different subtables, CBD can partition the large table into exactly the same resulting tables as the PBD.

We next discuss how we practically cut the graph into components. First, the problem of partitioning a graph into two equal-size components and minimizing the weights of edges among these components is known to be NP-hard [127]. Our problem is even more general, in the sense that in some cases we need to consider the sum of weights (for edges in the cut that are destined for different nodes) and in some cases we need to consider the product of weights (for edges that are destined for the same nodes). Moreover, we may want to first expand some of the rules.

In practice, we propose a greedy algorithm that solves this problem iteratively. At each iteration, we first try to partition the dependency graph into \( w \) equal-sized components, and minimize the weights of edges among these components. For this task, we use METIS [127], a tool to approximately partition graphs. Then, given the resulting partitioning, we evaluate it, and decide whether to expand one of the rules, and then go to the next iteration, or to finish by breaking all cut-edges. For instance, the wavy line in Fig. 8.3 depicts the cut of the dependency graph. After adding a default rule to each subtable, it results into two subtables of size 4 rules each (comparing to size 4 and size 5 in Example 13).

The decision whether to expand one of the rules or to finish by breaking all cut-edges depends on the quality of the partitioning. Namely, in case that the total weight of the edges in the cut exceeds some parameter \( w_0 \), we look at the destinations of the edges in the cut, and pick a vertex whose sum of incoming cut-edges’ weight is the largest. Then, we expand the rule corresponding to this vertex, thus eliminating all dependencies. Furthermore, since METIS does not necessarily return a perfectly balanced partition, we look at the ratio between the size of the largest component and the target size (that is, if a perfectly balanced partition would have been produced). If this ratio exceeds a certain parameter \( r_0 \) (namely, the partition is poorly balanced), we will try to break the largest component by expanding a rule within that component. Naturally, a good candidate for such an expansion is a rule that has few \( * \) bits (e.g., up to three) and many intra-component incoming edges (e.g., the one with the largest number of incoming edges).

**Theorem 45** The CBD algorithm stops.

**Proof.** At each iteration, one rule is expanded using at least one bit, that is, in the resulting table there is at least one \( * \) less. A table with not even one \( * \) has a corresponding dependency graph with no edges. Therefore, it can be easily partitioned into \( w \) equally-sized partitions with cut edge of weight 0. \( \square \)

### 8.4 The Rainbow Path Coloring Problem

After showing how to decompose the initial table into subtables, we now turn to show how to spread the subtables in the network.
Figure 8.4: Illustration of the model. Network $G = \langle V, E \rangle$ has vertex set $V = \{v_1, \ldots, v_4\}$ for the switches, and an edge set $E = \{e_1, \ldots, e_5\}$ for the links. There are three paths in the path set $P = \{p_1, p_2, p_3\}$, and, for example, $S(p_1) = \{v_1, v_2\}$ and $L(p_1) = \{e_1\}$. Finally, $G|_P$ is the same as $G$ but without the link $e_3$, which does not belong to any of the paths.

We model the network as a directed graph $G = \langle V, E \rangle$ with a vertex set $V$ and an edge set $E$, where $V = \{v_1, \ldots, v_{n_s}\}$ represents the set of $n_s$ switches and $E = \{e_1, \ldots, e_{m_l}\}$ the set of $m_l$ links. As a first step, we consider an homogeneous network where all switches are identical, and therefore have identical constraints on the table size. We will relax this assumption in Section 8.5.

Let $P = \{p_1, \ldots, p_f\}$ be the set of all flow paths in the network. For each flow path $p_i$, $(p_i)$ denotes its set of switches and $L(p_i)$ denotes its set of links.

Finally, given a graph $G$ and a set of paths $P$, we denote by $G|_P$ the projection of $G$ over $P$, namely $G|_P = \left( \bigcup_{p \in P} S(p), \bigcup_{p \in P} L(p) \right)$ is the subgraph of $G$ that contains only the switches and links that belong to at least one path in $P$. Fig. 8.4 illustrates the above definitions.

Recall that we aim to maintain full coverage of the original table semantics when dividing the work among switches. Informally, we would want to color each switch in one of $w$ possible colors (or with no color), subject to the constraint that each path must contain all $w$ colors. This would help us to divide the table among switches, so that each switch would need approximately $\frac{1}{w}$ of the entire table. Formally, the problem is defined as follows.

**Definition 13** Given a network $G = \langle V, E \rangle$, a flow path set $P$, and a number of colors $w$, the $(G, P, w)$ RAINBOW PATH PROBLEM is to decide whether there exists an assignment $\lambda : V \rightarrow \{\bot, 1, \ldots, w\}$, where $\bot$ corresponds to no color, such that each path $p \in P$ has at least one node of each color.\(^2\)

\(^2\)This last condition on the paths can be formally written as follows. Let $\lambda : 2^V \rightarrow 2^{\{\bot, 1, \ldots, w\}}$ denote the extension of $\lambda$ to a set of nodes: $\lambda(V') = \{w \mid v \in V', \lambda(v) = w\}$. A valid assignment $\lambda$ implies that for all
While the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM is defined as a decision problem, in practice, our goal is to maximize \( w \). Notice that the number of colors \( w \) is clearly at most the length of the shortest path in \( P \), which is in turn bounded by \( n_s \). Thus, the maximum number of colors can be obtained by applying the decision problem with any number of colors \( w \) up to the shortest relevant path size.

We now show that for general graphs, the decision problem is \( NP\text{-}hard \) even for two colors. The proof is based on reducing the 3-SAT problem \([128]\) to the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM.

**Theorem 46** Given a general network \( G \), a path set \( P \), and a number of colors \( w \), the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM is \( NP\text{-}hard \).

**Proof.** Let the Boolean expression \( B \) denote an instance of the 3-SAT problem with variables \( X_1, X_2, \ldots, X_{n_s} \). Thus, \( B = C_1 \land C_2 \land \ldots \land C_k \), where \( C_i = (X_i^1 \lor X_i^2 \lor X_i^3) \). We have to construct in polynomial time an instance of the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM so that it would have a proper coloring if and only if the Boolean expression \( B \) is satisfiable.

In practice, our proof needs two distinct constructions, although with a slight variation. One construction holds for the case where there exists an assignment \( A \) such that \( B \) is satisfiable with \( A(X_i) = 1 \), while the other construction holds for the case where \( B \) is satisfiable with \( A(X_i) = 0 \). Since there is only a slight variation between the two constructions, we focus on the first case. An explanation on how to apply the proof with the slight variation is found in the end of this proof.

For each variable \( X_i \), we allocate two distinct switches \( v_i \) and \( v'_i \), where \( v_i \) represents \( X_i \) and \( v'_i \) represents \( \neg X_i \). Furthermore, let \( M(X_i^j) \) be the switch representing \( j \)-th literal in the \( i \)-th clause. Finally, we add an additional baseline switch, denoted \( V^a \). So, that the constructed switch set \( V \) is

\[
V = \left( \bigcup_{i=1}^{n_s} \{v_i, v'_i\} \right) \cup \{v^a\}.
\]

We now skip to the construction of the path set \( P \). First, for each variable \( X_i \) we construct the path \( \langle v_i, v'_i \rangle \). There are \( n_s \) such paths in total. Then, for each clause \( C_i \), we construct the path \( \langle v^a, M(X_i^1), M(X_i^2), M(X_i^3) \rangle \). For example, for the clause \( (X_2 \lor \neg X_4 \lor \neg X_5) \) we construct the path \( \langle v^a, v_2, v'_4, v'_5 \rangle \). Finally, we add the path \( \langle v^a, v_1 \rangle \) to the path set. So the constructed path set \( P \) is

\[
P = \left( \bigcup_{i=1}^{n_s} \{(v_i, v'_i)\} \right) \cup \left( \bigcup_{i=1}^{k} \{(v^a, M(X_i^1), M(X_i^2), M(X_i^3))\} \right) \cup \{(v^a, v_1)\}.
\]

paths \( p \in P, \{1, \ldots, w\} \subseteq \lambda(S(p)) \).
The construction of the link set $E$ is easily deduced from the construction of the path set $P$, where there is an edge between two vertices if and only if this edge is within one of the paths.

Overall, we construct $2 \cdot n_s + 1$ switches and $k + n_s + 1$ paths. Therefore, the reduction is clearly polynomial.

Let $w = 2$, and $G = (V, E)$. The Boolean expression $B$ is satisfiable with $X_1 = 1$ if and only if there is a proper coloring that solves the constructed instance of the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM.

Given an assignment $A$ (with $A(X_1) = 1$) that satisfies $B$, we color the switches in $V$ as follows ($i \in \{1, \ldots, n_s\}$): If $A(X_i) = 1$ then $\lambda(v_i) = 2$ and $\lambda(v'_i) = 1$, while in the case where $A(X_i) = 0$, $\lambda(v_i) = 1$ and $\lambda(v'_i) = 2$. $\lambda(v^a)$ is set to 1. This implies that for all $i \in \{1, \ldots, n_s\}$, $\lambda(v_i) \neq \lambda(v'_i)$, and thus there are 2 distinct colors in the first $n_s$ paths in $P$. Since $A$ satisfies $B$, then for all $i \in \{1, \ldots, k\}$ it satisfies the clause $C_i$. It implies that at least one of the literal in $C_i$ equals 1. Therefore, the corresponding path in $P$ has the color 2 in addition to $v^a = 1$. The last path, that is, $\langle v^a, v_1 \rangle$, is easy to verify.

On the other hand, assume an assignment of colors $\lambda$ to the switches in $V$ that satisfy the condition of $w$ colors per path. Without loss of generality, due to the last path in $P$, we assume that $\lambda(v^a) = 1$, and $\lambda(v_1) = 2$. We build the following variable assignment $A$: $A(X_i) = \lambda(v_i) - 1$. First, we get that $A(X_1) = 1$. Second, from the definition of the construction, for all $i \in \{1, \ldots, k\}$, the clause $C_i$ is satisfied at least by one literal (whose corresponding switch color is 2).

Finally, we may repeat the same proof with a slight variation in the last path of the path set. Instead of $\langle v^a, v_1 \rangle$, we add to the path set $P$ the path $\{v^a, v'_1\}$. Following this variation, it is possible to prove that the Boolean expression $B$ is satisfiable with $v_1 = 0$ if and only if there is a proper coloring that solves the constructed instance of the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM, with $w = 2$.

Having covered the two cases, we get the claimed result.

We now obtain efficient solutions for special graph topologies, as well as heuristics for the general case.

### 8.4.1 The Rainbow Path Coloring in Trees

Nowadays, OpenFlow-equipped networks are often deployed in data centers, where the network itself has a regular structure. A prime example of such a topology is a tree: a fully connected graph that has no cycles.\(^3\)

In some cases, there is also a restriction on the relevant paths that should be considered; e.g., assuming all paths originate from or are destined to a single node.

In this section, we first tackle these single-source (equivalently, single-sink) trees. Our results are slightly more general, as they require that only the projection of the paths on

---

\(^{3}\)Some datacenters have a fat-tree topology. Extending our results to fat-tree is part of our future research.
the original topology is a tree. For example, if all paths are the shortest possible, a single-source (single-sink) setting always forms a tree (a.k.a. the shortest-paths tree). We show a simple valid coloring of size \(s + 1\), where \(s\) is the shortest path size. In this coloring, the color is simply given by the distance to the single ingress (or egress) switch.

**Theorem 47** Given a network \(G\), and a flow path set \(P\) such that all paths originate from or are destined to a single node and follow the shortest-path scheme, then there is a valid color assignment \(\lambda\) with \(s + 1\) colors to the \((G,P,w)\) Rainbow Path Problem, where \(s\) is the shortest path size in \(P\). Furthermore, \(\lambda\) can be computed in \(O(m_l + n_s)\) time.

**Proof.** Without loss of generality, assume that all paths in \(P\) originate from a single source \(x\), and denote by \(d(x,y)\) the distance in edges between node \(x\) and some node \(y\). Let \(\lambda\) be the following assignment function: for each \(y \in \bigcup_{p \in P} S(p)\),

\[
\lambda(y) = \begin{cases} 
    d(x,y) + 1 & d(x,y) \leq s \\
    \perp & \text{otherwise}
\end{cases}
\]

where \(\perp\) indicates that no color is given to the node, and therefore, no table should be installed at that node. Notice that \(\lambda\) can be computed in \(O(m_l + n_s)\) time using the Breadth-first search (BFS) algorithm [129] originating from \(x\).

Consider a path \(p \in P\), which starts at \(x\) and follows a shortest-path scheme. Thus, \(S(p)\) contains nodes of increasing distances until the length of \(p\), which is at least \(s\). Thus, \(\lambda(S(p)) = \{1, \ldots, s + 1\}\), and the claim follows. \(\Box\)

Next, we deal with the more general case in which \(G|_P\) is a tree. In such a case, we show a valid coloring with \(\left\lceil \frac{s}{2} + 1 \right\rceil\) colors, where \(s\) is the shortest path size in \(P\).

**Theorem 48** Given a network \(G\) and a flow path set \(P\) with a shortest path of length \(s\), if \(G|_P\) is a tree, then there is a valid color assignment with \(\left\lceil \frac{s}{2} + 1 \right\rceil\) colors to the \((G,P,w)\) Rainbow Path Problem. It can be computed in \(O(m_l + n_s)\) time.

**Proof.** Let \(w = \left\lceil \frac{s}{2} + 1 \right\rceil\), and pick an arbitrary node \(x \in S(P)\). Consider the following color assignment \(\lambda\):

\[
\lambda(y) = (d(x,y) - 1 \mod w) + 1.
\]

Note that, by definition, the values of \(\lambda\) are in \(\{1, \ldots, w\}\). \(\lambda\) can be computed using a BFS from node \(x\) in \(O(n_s + m_l)\) time.

We next show that each path contains all the colors. Consider a path \(p \in P\), and let \(y \in S(p)\) be the node with minimal distance to \(x\). Since \(G|_P\) is a tree, for each other node \(y' \in S(p)\), \(d(y',x) = d(x,y) + d(y,y')\) (otherwise, there is a path between \(y'\) and \(x\) that does not go through \(y\), implying that there is a cycle in \(G|_P\)). Let \(p_1, p_2\) be the division of path \(p\) into two paths: \(p_1\) starts in the first node of path \(p\) and ends in \(y\), \(p_2\) starts in \(y\) and ends in the last node of path \(p\). Without loss of generality, assume that \(p_1\) is
longer than \( p_2 \); thus the length of \( p_1 \) is at least \( \lceil \frac{s}{2} \rceil \) edges, implying that \( |S(p_1)| \geq \lceil \frac{s}{2} + 1 \rceil \). Furthermore, since there is only a single simple path between each two nodes in the tree, the set of distances between nodes in \( S(p_1) \) and \( y \) is \( \{0, \ldots, |S(p_1)|\} \), immediately implying that \( |\lambda(S(p_1))| = \{1, \ldots, |S(p_1)|\} \), and the claim follows.

Another special case where all paths are of length 2 is considered in the appendix.

### 8.4.2 The Rainbow Path Coloring With All Paths Of Length 2

In this section we consider the case where all paths in \( P \) are of size 2. In this case, the problem is not NP-hard, and in fact can be solved in linear time. This is reflected in the following theorem.

**Theorem 49** Given a general network \( G \), a path set \( P \) with paths of size 2, and \( w = 2 \), the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM can be solved in \( O(|V'| + |E'|) \) time, where \( V' \) and \( E' \) are the switch and link sets of the projection \( G|_P \) of \( G \) over \( P \).

**Proof.** If all paths are of size 2, it clearly follows that there exists a valid coloring if and only if \( G|_P \) is a bipartite graph. To determine whether a given graph is bipartite, we simply use the Breadth-first search (BFS) algorithm [129], whose complexity is \( O(|V'| + |E'|) \). \( \square \)

### 8.4.3 The Rainbow Path Coloring in General Graphs

Since the \( \langle G, P, w \rangle \) RAINBOW PATH PROBLEM is not tractable in general graphs, we present a greedy heuristic that might yield a suboptimal solution. Yet, our simulations show that in practice the margin of error is on average within 2\% (see Section 8.6).

More specifically, our GREEDY algorithm (see Algorithm 1 for pseudo-code) works in iterations. At each iteration, which corresponds to a new color, GREEDY continuously picks uncolored nodes one by one, until each path contains at least one of the picked nodes in this iteration. In such a case, the nodes picked are colored with a new color, and the algorithm continues to the next iteration. If at some iteration, even after picking all uncolored nodes, there is at least one path that does not contain any of the picked nodes, then those nodes remain uncolored (\( \bot \)), and the algorithm stops. Note that, in any case, the algorithm never stops in the first iteration, that is, it always succeeds to color the nodes using at least one color.

We next present two variants of this algorithm, which differ in the way the nodes are selected at each iteration. In the first variant, which we call 1-GREEDY, at each choice, we pick the node \( v \) that maximizes the number of paths that contain \( v \) but do not contain the new color.

The following theorem captures the time complexity of this algorithm:

**Theorem 50** 1-GREEDY runs in \( O(n_s^2 \cdot f) \) time complexity, where \( n_s \) is the number of nodes and \( f \) is the number of paths.
**Algorithm 1**: Pseudo-code for the $q$-GREEDY algorithm

- **Input**: node set $V$, path set $P$
- **Output**: a valid color assignment $\lambda$, number of colors $w$
- $w = 0$;
- set $\lambda(v) = \bot$ for all $v \in V$;
- while $V \neq \emptyset$ do
  - $P' = P$; $V' = \emptyset$;
  - $\text{Search} = \text{TRUE}$;
  - while $\text{Search}$ do
    - $V^0 = \arg\max_{V' \subseteq V, |V'| \leq q} \left\{ \left\{ p | V \cap S(p) \neq \emptyset, p \in P' \right\} \right\}$;
    - $P^0 = \left\{ p | V \cap S(p) \neq \emptyset, p \in P' \right\}$;
    - $V = V \setminus V^0$; $V' = V' \cup \{V^0\}$; $P' = P' \setminus P^0$;
    - $\text{Search} = (P' \neq \emptyset)$ and $(V^0 \neq \emptyset)$;
  - if $P' = \emptyset$ then
    - $w = w + 1$;
    - set $\lambda(v) = w$ for all $v \in V'$;
    - $V = V \setminus V'$;
  - else
    - $V = \emptyset$;

**Proof.** When we pick a node to color, we first need to consider all other remaining nodes to ensure that this node is the one which belongs to the largest number of paths; each such comparison takes at most $f$ steps (counting all the paths). Hence, the total number of steps required to choose all $n_s$ nodes, is $f \cdot ((n_s - 1) + (n_s - 2) + \ldots + 1) = O(n_s^2 \cdot f)$. 

The second variant, $q$-GREEDY, generalizes the 1-GREEDY algorithm by considering, at each step, a set of up to $q$ nodes (instead of a single node). It chooses the set of nodes that maximizes the number of paths for which there is at least one node in the set.

The next example shows an execution of 1-GREEDY and an execution of 2-GREEDY that differ in their outcome. This demonstrates the tradeoffs in fixing the parameter $q$.

**Example 14** Consider the example in Fig. 8.4. We first run 1-GREEDY. At the first iteration, nodes $v_1$, $v_2$, and $v_3$ belong to two paths; assume that 1-GREEDY first picks $v_1$ and colors it in the first color. Then, in order to color $p_3$, it picks $v_2$ and colors it in the first color as well. Note that all nodes of the first path $p_1$ are now colored, implying that any additional iteration will fail, resulting in a valid coloring of only one color.

In contrast, $q$-GREEDY with $q = 2$ first picks nodes $v_2$ and $v_3$, since all three paths traverse through either one of these nodes. Then, $v_1$ and $v_4$ can be colored with an additional color, resulting in a valid coloring with 2 colors.

The following theorem is a simple generalization of Theorem 50:
Theorem 51. \( q \)-greedy runs in \( O(n_{s}^{q+1} \cdot f) \) time complexity, where \( n_{s} \) is the switch set size and \( f \) is the path set size.

8.4.4 An Optimal Solver

Although the \( \langle G, P, w \rangle \) Rainbow Path Problem is NP-hard, it is possible to compute the optimal solution for small instances of the problem. In this section, we present an algorithm, based on dynamic-programming, that practically solves the \( \langle G, P, w \rangle \) Rainbow Path Problem with projected graphs of up to 17 nodes. Note that each (isomorphic) coloring corresponds to a specific partition of a set of \( n_{s} \) labeled elements [130]. The number of possible partitions of \( n_{s} \) elements is the \( n_{s} \)-th Bell number, whose value for \( n_{s} = 16 \) is approximately \( 10^{10} \). Thus, a naive approach of testing all possibilities is intractable in a reasonable time. The main motivation is two-folded. First, it may be satisfactory for some real-life instances of the \( \langle G, P, w \rangle \) Rainbow Path Problem. Second, this algorithm is used in Section 8.6 as a baseline for evaluating the performance of the 1-greedy and \( q \)-greedy.

Our algorithm works in two phases. In the first phase, we find a set \( S \) of all subsets of the switch set that have the following property: Given a subset \( q \in Q \), coloring all switches in \( q \) with a specific color results in coloring all paths in the path set. We also make sure that for all \( q \in Q \), the subset \( q \) is minimal in the sense that there is no subset of \( q \) that has the same property.

To find the set \( Q \), we start with a set of \( n_{s} \) singletons. Then, at each step, for each of the subsets so far, we try to add a new switch with index larger than the largest switch index of this subset. If adding a new switch makes the number of paths that are colored larger, then the new subset is saved for the next iteration, otherwise it is dismissed. Also, if the new subset covers all paths, then it is added to \( S \).

Next, after having computed the set \( Q \), we go to the second phase of our algorithm, where we find the maximum size set of disjoint subsets in \( Q \), whose size corresponds to the optimal number of colors that can be used in the original \( \langle G, P, w \rangle \) Rainbow Path Problem. We note that the second phase is essentially an instance of the maximum set packing problem, and can also be solved using a dynamic programming technique.

8.5 Multicolored Switches

Up until now, we have considered the \( \langle G, P, w \rangle \) Rainbow Path Problem where a single color is assigned to each switch. However, in practice, we may want to assign more than one color to each switch.

The motivation for such an assignment is three-folded. First, this would facilitate the implementation of a heterogeneous software-defined network, in which nodes with a larger-capacity table can be assigned multiple colors.

Second, there is additional degree of freedom in the problem, implying that there are more feasible colors assignments. To see this, consider Fig. 8.5, which shows a ring graph of
Figure 8.5: A network with no valid coloring to the \( (G, P, w) \) RAINBOW PATH PROBLEM with \( w = 2 \)

size 3 in which each path consists of a single edge. The only valid solution of the \( (G, P, w) \) RAINBOW PATH PROBLEM is when all switches are colored with the same color. However, the graph can be colored in three colors, when each switch is allowed to be colored with two colors (namely, \( \lambda(v_1) = \{1, 2\}; \lambda(v_2) = \{2, 3\}; \lambda(v_3) = \{3, 1\} \)). Specifically, we are interested in the ratio of the number of colors in each switch to the total number of colors. In this example, the ratio is \( \frac{2}{3} \), implying that each switch should hold approximately two thirds of the table (as opposed to the entire table in the \( (G, P, w) \) RAINBOW PATH PROBLEM solution).

The third motivation for assigning multiple colors to a node is in cases where the graph \( G \) contains only few very short paths, while other paths are relatively long. This is because in the \( (G, P, w) \) RAINBOW PATH PROBLEM, the length of the shortest path is an upper bound on the number of colors that can be used. Therefore, it also determines a lower bound on the size of the table stored in the switches. This implies that switches on longer paths, which can potentially share their table with more switches, will still need to store a large table. In that case, it is appealing to assign multiple colors to switches on shorter paths (resulting in relatively large tables), while the rest of the switches are assigned only with few colors, which correspond to smaller tables.

We now formally define the multicolored problem.

**Definition 14** Given a network \( G = (V, E) \), a set of paths \( P \), a number of colors \( w \), and for each node \( v_i \) a maximum number \( d_i \) of colors that it can accept, the \( (G, P, w, d) \) RAINBOW PATH PROBLEM is to decide whether there exists an assignment \( \lambda : V \rightarrow 2^{\{1, \ldots, w\}} \), such that each path \( p \in P \) has at least one node of each color, and no node \( v_i \) has more than \( d_i \) colors.\(^4\)

\(^4\)Formally, in this case, \( \bar{\lambda}(V') = \{w \mid v \in V', w \in \lambda(v)\} \). A valid assignment \( \lambda \) implies that for all paths \( p \in P, \{1, \ldots, w\} \subseteq \bar{\lambda}(S(p)) \) and for each \( v_i \in V, |\lambda(v_i)| \leq d_i \).

149
Note that the fraction of the original table that is stored in node $v_i$ is approximately $|\lambda(v_i)|/w$, assuming that the table decomposition algorithms (Section 8.3) are efficient.

Clearly, as the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM is NP-hard, so is the $\langle G, P, w, d \rangle$ RAINBOW PATH PROBLEM. However, we can reuse our results on the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM using the following reduction: Given the instance of the $\langle G, P, w, d \rangle$ RAINBOW PATH PROBLEM, we split each node $v_i$ into a chain of $d_i$ nodes, and make each path that goes through $v_i$ to go through the whole corresponding chain. Then, we get an instance of the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM that corresponds to the original $\langle G, P, w, d \rangle$ RAINBOW PATH PROBLEM. Note that in the $\langle G, P, w \rangle$ RAINBOW PATH PROBLEM, it is possible that a node remains uncolored. This corresponds to the case where, in the original problem, the number of colors assigned to some node $v_i$ is lower than $d_i$.

8.6 Experimental Results

We now turn to evaluating our algorithms. We first check the decomposition algorithms (Section 8.3) and then the table distribution algorithms (Sections 8.4–8.5).

8.6.1 Table Decomposition

We first consider the PBD and CBD algorithms for decomposing tables, as presented in Section 8.3.

We define the quality of a table decomposition algorithm as the ratio between the number of rules in the original table, and the product of the largest resulting subtable size by the number of subtables $w$. The quality is therefore between 0 and 1, where higher quality values implies a better decomposition. Specifically, a quality of 1 means that the largest subtable size has exactly an ideal fraction $1/w$ of the number of the original rules. Note that this quality can only be used to compare among different algorithms for the same value of $w$. Furthermore, it is most likely that the quality is decreased when $w$ is increased.

We compare PBD and CBD algorithms with a bit groups algorithm based on [15]. Through an exhaustive search, this algorithm selects the $\log_2 w$ pivot bits that maximize the quality. Thus, it only works for values of $w$ that are powers of 2.

Fig. 8.6 shows the quality of the three algorithms as the number $w$ of partitions grows. For the simulations, we have created 100 random logically-minimized rule-sets with 12 bits and 30 rules each. PBD slightly outperforms bit groups, except when $w = 2$, where they perform similarly. CBD clearly outperforms both over the entire range.

We also evaluated PBD and CBD with the twelve standard classification benchmark rule-sets of ClassBench [131, 123].

When the dependency graph of CBD is relatively sparse, PBD and CBD usually display a quality between 0.7 to 0.99 for various values of $w$. Note that for $w$ that is not a power of 2, the quality of PBD may drop significantly, while CBD remains stable. Such a case is shown in Fig. 8.7.
Figure 8.6: Evaluation of the quality metric of the PBD, CBD, and a non-iterative algorithm that selects all pivot bits at once [14, 15]. The input is a synthetic rule set.

Figure 8.7: Quality of partitioning of the PBD and CBD using a benchmark rule set

However, when the dependency graph of CBD is dense, CBD may show poor results. This is due to an instability of METIS; namely, expanding one rule in the input may result in significantly worse partitioning (where we would expect to have a partitioning at least as good as before the rule expansion). In future, we aim to overcome this problem by implementing our own graph partitioning mechanism.

8.6.2 Table Distribution

In this section, we evaluate the greedy algorithms for the single-color case, as introduced in Section 8.4.

To analyze their performance, we produce random instances of the \((G, P, w)\) RAINBOW PATH PROBLEM in the following manner: given a number of switches \(n_s\) and a number of paths \(f\), we add each switch to each path with probability \(p_n\) independently of the other switches or paths. Note that for a given path, the actual order of switches within the path and how they are connected to each other (namely, the exact network topology) is irrelevant to the \((G, P, w)\) RAINBOW PATH PROBLEM. The number of switches in each path of our instances follows a Binomial distribution with parameters \(n_s\) and \(p_n\). Also note that the
Figure 8.8: Evaluation of 1-GREEDY, 2-GREEDY, and 3-GREEDY, for various values of $n_s$, $f$, and $P_n$. 

(a) $n_s = 10$, $f = 10$, $p_n = 0.8$

(b) $n_s = 20$, $f = 10$, $p_n = 0.4$
length of the shortest path is an upper bound on the size of a valid coloring. We note that these random instances assume, among other things, independence between paths, which is not the case in real-life networks. Our future research includes evaluating the networks under real-life network topologies.

Fig. 8.8 shows the average size of the valid coloring obtained by our 1-greedy, 2-greedy, and 3-greedy algorithms, as a function of the shortest path in the random instance of the problem. We ran 10,000 random instances of the problem. Fig. 8.8(a) illustrates the average size of the valid coloring size obtained by our algorithms for $n_s = 10$, $f = 10$, and $p_n = 0.8$, while Fig. 8.8(b) shows it for $n_s = 20$, $f = 10$, and $p_n = 0.4$. Note that for a shortest path size 5 and $n_s = 10$ switches, the maximum valid coloring size is 5, and indeed, our algorithms achieve on average a valid coloring of size 4.3043, 4.7572 and 4.7573 for $q = 1$, $q = 2$, and $q = 3$, respectively. Clearly, a larger value of $q$ results in a larger number of colors, on average.

We further study the our algorithms on smaller networks, where we are able to compute the optimal valid coloring size.

Fig. 8.9 shows the number of colors found by the greedy approach in terms of percentage of the optimal solution. The parameters in this case are $n_s = 7$, $f = 7$ and $p_n = \frac{5}{7}$, and we ran the simulation 1000 times. Our results yield that, in these cases, that the greedy approach finds a valid coloring whose size exceeds (on average) 98% of the optimal solution.

### 8.6.3 Greedy Approach Evaluation - Multiple Color case

In this section we evaluate our solution to the $(G, P, w, d)$ rainbow path problem. To compare the algorithm performance with various values of $d$, we normalize the valid coloring size found by $d$. This corresponds to the share of the work-load that each switch gets in the worst case.

Fig. 8.10 shows the normalized valid coloring size as a function of the shortest path
in random instances of the problem. We ran 10,000 random instances of the problem. In Fig. 8.10(a) the parameters are $n_s = 10$, $f = 10$ and $p_n = 0.8$, while in Fig. 8.10(b) they are $n_s = 20$, $f = 10$ and $p_n = 0.4$.

The simulation results show that as $d$ increases, the normalized number of colors also increases. For example, for a shortest path size $s = 5$ and $n_s = 10$ switches, a normalized valid coloring of size 4.3457, 4.5429 and 4.6101 was achieved for $d = 1$, $d = 2$ and $d = 3$, respectively.
Chapter 9

Conclusion and Future Work

Hash tables and other tables in general have become crucial algorithmic building blocks for contemporary networking devices that handle and analyze large amounts of data at very high speeds. For instance, for each arriving packet, routers need to perform an address lookup and several flow classification and identification operations, each often relying on a hash table scheme. In this dissertation, we considered several scenarios and memory models.

First, we considered multiple-choice hashing schemes with a uniform memory model and an overflow list, implemented using a CAM chip.

In Chapter 4, we considered multiple-choice hashing schemes without deletion operations, i.e. only insertion and query operations. We were able to find the expected overflow fraction of several schemes, and to show that they are optimal with respect to the average number of memory accesses.

Then, Chapter 5 overviews our results in cases where deletions are also allowed. In particular, we demonstrated that, when the memory is bounded, dynamic schemes behave significantly worse than their static counterparts. This decrease in performance is inherent to the problem, as shown by our lower bounds.

Moreover, we considered two hashing schemes that we proved to be optimal: a single-choice hashing scheme that was used to demonstrate our approach and techniques, and a multiple-choice scheme that inserts the elements greedily.

However, due to the slow decrease of the lower bound, optimality may be insufficient for certain applications. Therefore, we suggested moving back elements from the overflow list as soon as a deletion occurs. We have shown through simulations that this strategy beats the lower bound of the dynamic case (where moving back elements is not allowed).

In Chapter 6 we presented an access-efficient variant of Bloom filters for networking devices, called the Balanced Bloom filter. In this variant, each element is first assigned to a memory block, where a local Bloom filter is maintained. Our basic approach was twofolded. First, we proposed to maintain load-balancing schemes: a simple approach (using only a single hash function), a sequential approach, and a mutli-level hash-table approach.
And second, we proposed to use an overflow list that can store elements that are hashed to overloaded buckets.

To study this problem, we presented an access-constrained balancing problem and showed that, for some specific parameters, our proposed load-balancing schemes are optimal.

Since the cost function is very general, the balancing problem can be used in other contexts. We demonstrated how it can be used in order to find optimal balancing schemes that take into account the variance of the query-time and not just its expected or worst-case time.

Next, we considered in Chapter 7 a combined memory model (e.g., of both SRAM and DRAM), in which some of the elements are stored in a fast memory, while others are stored in much slower memory.

We set 2 choices per element in the SRAM chip, and suggested that the largest number of elements possible are stored in this memory. For that, we provided an exact expression for the expected maximum matching size of a random bipartite graph with each left-side vertex picking $d = 2$ right-side vertices, for any number of left-side and right-side vertices. Then, we deduced asymptotic results as the memory size goes to infinity. Both results serve as upper bounds for any multiple-choice hashing algorithm with $d = 2$ choices. Thus, introducing a capacity region for these schemes. We further analyzed several hashing variants, in which the memory is statically partitioned, we have more than two hash functions or we have (on average) less than two functions. Our results illustrate the impact of the SRAM/DRAM access time ratio on the parameters choice. In particular, we show that the common intuition of avoiding DRAM accesses by using highly efficient schemes is not always correct.

We are currently interested in the following two open problem. First, we believe that it is important to find the average maximum matching size for any arbitrary $d$, and more generally any vertex number distribution. While we have been able to obtain closed bounds using the technique provided in this chapter, we have not found general exact results. And second, we believe that cases where elements can be split between two or more bins are also interesting. Such cases are common for example for security and robustness purposes.

Last, in Chapter 8, we show how network rules usually implemented using TCAMs at the ingress points of the network can be split among all routers in the network, such that each switch need to maintain only a small TCAM. In particular, we proposed Palette, a framework to decompose and distribute SDN tables across the network. Palette is especially important as switch table sizes can become a bottleneck in scaling SDNs. Moreover, it facilitates handling the heterogeneity of switches in the network and the changes of equipment.

We modeled the problem in a graph-theoretic manner, and proposed several algorithms, both for decomposing one table to semantically-equivalent subtables and for spreading these subtables across the network. Our algorithms were evaluated both under random and real-life instances.
As future work, we now plan to implement Palette over OpenFlow controllers, thus providing an automatic tool to decompose and distribute SDN tables. We note that a major challenge in OpenFlow implementations is OpenFlow’s restrictions on the structure of patterns in the table (in OpenFlow 1.0, two fields are allowed to be prefixes, and the other fields can be either exact or entirely *-bits [120]); this will require adaptation of our decomposition algorithm accordingly. Finally, we plan to extend our optimal coloring algorithms to handle additional network topologies (e.g., fat trees, which are common in contemporary datacenters).

In addition, there are several interesting variants of this problem. One such variant is the case where the rules are used for measurement of traffic. In such a case, additional constraints need to be added, for example, a constraint that each color is traversed exactly once in each of the paths (this corresponds to not measuring any packet twice). While this constraint may decrease dramatically the degree of freedom, we believe that it can be relaxed. One direction may be then to define the sufficient and necessary conditions for this scenario.

Another interesting variant is to consider a robust network that can overcome a failure of up to one of the TCAMs and preserve the semantics and the global operation in such a case. This necessitates redundancy of the rules across the network. One of the goals, for example, is to minimize this redundancy.
Bibliography


159


161


הגישה שטורגנית לעבורה 3 מבסס על פתרונות שונים לתתי בעיות. היא מתאימה ל몇 טבלאות באמצעות אלגוריתמים
(כגון מומג), כדי להפוך את הפתרון לעובדה הוא חוק מתאימה בצפיפות המקורית של החוק המתאימה לתה, כל אחד
れます לפי התו אחרון, כאשר חוק מתאימה בצפיפות המקורית של החוק מתאימה
בטבלאות מתאימות. בשתי התו-תה, ללא בחירה העדשה של החוק מתאימה בצפיפות המקורית של החוק מתאימה
בכת santa מתאימות. ניתן לנצל את האפשרויות של החוק העדשה כדי לשייך את כל ליניות
יפורסמי לפרסום. התו העדשה השניה, היא פורח מתאימות בין נתיבי השתייה לשתי התו
בטבלאות שימשו כשל הכלי בפרסום בין לטבלאות המתאימות, בכל התו
בטבלאות אופטימליי ל��ם שטח. לפי התו, בטבלאות כשל העדשה של החוק
יפורסמי.
морיט המחקק היא להבינהображен אשר מתמחום לאלים הרצות וفق סנגור. בʷפך, מתוכך שללא המסבאות אושם מתמחום לאלים, לגופה בשיעי פתרונות בbeer חיים וברכיבת בשיעי פתרונות בbeer חיים.

למסיים חיכרה.

הפרמות האשרים הם מחקים הבנויים הממוצעים, המבינים על ידי מספים הגישות לזכרון אטוי בוחר

במערך המבנה עבורה. מודרני המרחב שבוי זה החוק בתבניות ביןansom פvoor, ובישה

למספי ריכב זכר בפיוספים אחר, ובישה בועת בעבורה הממוצעים (ולשון השיעי פקורית

עבורה). הניב כי מחקים המרחב שבוי 보וץ קרבה במיתות לاذיבת הנושאים רכיבי, או שוא

涂抹 למחרות ממלאות סמוכות בברח המרחב שבוי לזרע פפורי פפוריotecעבורה.

הפרמות השתי אכזב זכר המרחב למחרות השפה. broader כלא תיעדו כי זכר המרחב המובול

ביותר, על פי המבודד המפותל לשון CAM-B או CAM-C, זכר המרחב שאין רכיבי איו היווה את מר

הבישוע עבורה.

הפרמות האשרים הם מחקים הבנויים הממוצעים, המבינים על ידי מספים הגישות לזכרון אטוי בוחר

לملابس זכר.זוגת מחקים הבנויים הממוצעים, או מספים הגישות לזכרון אטוי בוחר, הם אטוי בוחר.

ל,'%ץ CAM-C זכר המרחב שבוי בברח עבורה, או מספים הגישות לזכרון אטוי בוחר, או

 parfait הзамен הзамен אחר לשון פפורי פפוריotecעבורה.

הפרמות השתי אכזב זכר המרחב למחרות השפה. broader כלא תיעדו כי זכר המרחב המובול

ביותר, על פי המבודד המפותל לשון CAM-B או CAM-C, זכר המרחב שאין רכיבי איו היווה את מר

הבישוע עבורה.

למסיים חיכרה.

הפרמות השתי אכזב זכר המרחב למחרות השפה. broader כלא תיעדו כי זכר המרחב המובול

ביותר, על פי המבודד המפותל לשון CAM-B או CAM-C, זכר המרחב שאין רכיבי איו היווה את מר

הבישוע עבורה.

למסיים חיכרה.

הפרמות השתי אכזב זכר המרחב למחרות השפה. broader כלא תיעדו כי זכר המרחב המובול

ביותר, על פי המבודד המפותל לשון CAM-B או CAM-C, זכר המרחב שאין רכיבי איו היווה את מר

הבישוע עבורה.

למסיים חיכרה.
תקציר

מתוך התוכניות היא נוכחות אולטרה נראית לדוגמה צמחים בצבעים שונים. המISOString של מית资源配置ים.

הקביצה היא 실ולית במובנה של תכונה יריעה. הוא מתאם מתגים שהם מתאימים אשר נתונים

מידע המטגים המבוססים על סerialization, ת自卑,字体, כדי לאפשר את התפקיד.

עדות תכונה בהם משתמש, ניתן перемיק את המיקי מתאימה לביצוע כל פעולות אך גם работу ב faker, הצגת תמונות

ובבכל העבורה ובבקרת מתג

כותרת

בחינת שאלים בגישה, לדוגמה, ב القضية עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ𡎴יות שזור.

ב䷣ לעבורה裔 אבּאפּ או יישום במרבית הקטן ייחודיstoff CAM עם תכונה זו

 mejorar את המוסכם, ב 있지만 ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה לברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב מקרה עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב䷣ לעבורה裔 אבּאפּ או יישום בHomeAsUpEnabled קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה לברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב מקרה עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה לברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה לברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נ众所יות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נсорיות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, לדוגמה, בخصوص

תכלית שאלים בגישה, לדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נсорיות שזור.

ב瓤 לעבורה裔 אבּאפּ או יישום בですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, Bþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, ב случай עבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, Bþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, ב случай Unבורה.

מעורבות טבלאות מתאימה, או ריבים אולטרה נсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, Bþึו, ידועים, הרשתות, אוניהים Wחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, ב случай Unבורה.

מעורבות טבלאות מתאימה, או ריבים Aולטרה Nсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, Bþึו, ידועים, הרשתות, אוניהים Wחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, Bخصوص

תכלית שאלים בגישה, Lלדוגמה, B случай Unבורה.

מעורבות טבלאות מתאימה, או ריבים Aולטרה Nсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים וחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, B случай Unבורה.

מעורבות טבלאות מתאימה, או ריבים Aולטרה Nсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים Wחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, Bخصوص

תכלית שאלים בגישה, Lלדוגמה, B случай Unבורה.

מעורבות טבלאות מתאימה, או ריבים Aולטרה Nсорיות שזור.

B瓤 לעבורה裔 אבּאפּ או יישום Bですから קטן ייחודיstoff CAM עם תכונה זו

improve את המוסכם, בþึו, ידועים, הרשתות, אוניהים Wחברת

תחביב עסוך טלבהולם אשר מתאימה Lברך.

ואשר בשיתוף פעולה עם עבור עבורה裔, Lלדוגמה, בخصوص

תכלית שאלים בגישה, Lלדוגמה, B случай Unבora.
הсетת תודה

ברצוני להודות לנציגים אלה, צ'ק קסלסי, ווד ויו, על הניחים בהן אימוץ והמתארגנויות עקב כך. ב都被ל על הדרכה על תחתית במעון מחקרים, ובמיזוג על הדרכה באים בין מוכנים ייחודיים. מעברقوا לכל שדה תחום למידת המשך, א ספרות מבית שנותיה: תמצית והשיבת א実際にים מתוך התこれまでות.

בנוסף להישגיו הרגשיים, זכה ה.Conference,רצוני להודות למדעי המחשב ועל תמקומי הרחב לאובדות ו(changes, שהביאו לקידום, ואזרחי

ortic ה.Conference. ש Guinea, שום דרג מהלא כן קוראים בלעד כן.欧בדו זה מצותיכים loans.

אני מודה לכולכם על הע listView כנספי המבחין בהשתל bilder: ה.Conference.

המחקק עשה בחינתית ו文章来源セット קסלסי ווד ויו בפקולות למOfString המ médec.
הלוות מחירות בשכני רשת

תובור על מתקד

לשם مليולי חלב של הדורשות שבבלת תואר
דוקטור לפילוסופיה

יוסי קנייז

הוגש לפננ הרכה – מכון טכנולוגי לישראל

ינואר 2014
החלשות מהירות ולהקיני ראש

יוסי קנייזו