Bachelor's Thesis

A Compact Cache-Efficient Function Store with Constant Evaluation Time

Wei Zhou

Submission Date: 31 October 2013

Supervisors: Prof. Dr. rer. nat. Peter Sanders
Dip.-Inf. Robert Schulze
Dipl.-Inform., Ingo Müller, M.Sc.

Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 31. Oktober 2013

Wei Zhou

Abstract

A new data structure to store a set of key-value mappings for finite static key sets is presented. The data structure, which is called Cache-Efficient Function Stores (CEFS), can be built in linear expected time and supports evaluation for a key within worst-case constant time. Furthermore, (i) the building process can be parallelized to achieve massive speed-up over known methods; (ii) an evaluation needs less than two cache misses in average case for many applications, improving upon all known methods. The data structure is also compact, needing only $\mathcal{O}(n)$ bits extra space to be stored. The data structure is flexible in that there are many parameters that can be configured in order to fit in specific applications. The time and space properties for different parameters can be predicted to great precision in advance with formulae developed in the thesis. It is also possible to automate the selection of parameters. Experiments have shown the efficiency of the new data structure and confirmed the theoretical analysis.

Acknowledgments

The author would like to thank Prof. Peter Sanders at the Karlsruhe Institute of Technology, from whom the primary idea of the data structure originates, as well as Ingo Müller from the Karlsruhe Institute of Technology and Robert Schulze from SAP AG for their kind guidance and all the valuable discussions.

Contents

List of Figures

List of Tables

List of Algorithms

List of Notations

$[a..b]$	The set of all integers in the closed interval $[a, b]$, i.e. $\{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$.
$A[i]$	The i-th element of sequence A. A can also be any multiset, in which case $A[i]$ means the i-th element in a *fixed* permutation of all elements in A. $i \in [0..\lvert A\rvert - 1]$.
$A[a..b]$	An array or a sequence with indices from a to b or the sub-array of A consisting of the elements $A[a], A[a+1], \ldots, A[b]$, depending on the context.
$\lfloor x \rfloor$	The biggest integer that is smaller than or equal to x.
$\lceil x \rceil$	The smallest integer that is larger than or equal to x.
$\Pr[A]$	The probability that event A occurs.
$\Pr[A \mid B]$	The probability that event A occurs under the condition that B occurs.
$\mathrm{E}[X]$	The expected value of the random variable X.
$\binom{n}{r}$	The binomial coefficient, which is the number of ways to choose an r-element subset from a set of n distinguishable elements.
$\mathrm{Pois}(\lambda)$	The Poisson distribution with parameter λ. The probability mass function is $\Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$.
$\mathrm{B}(n, p)$	The binomial distribution with parameter n and p. The probability mass function is $\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$.
$n!$	The factorial of n, defined as $n! := \prod_{i=1}^{n} i$.
$\left\{ {n \atop r} \right\}_2$	The 2-associated Stirling numbers of the second kind, which count the ways to partition a set of n labeled objects into r unlabeled boxes such that every box contains at least two objects [1, 2, 3].
$a \mid b$	“a divides b”, which means there exists an integer $q \neq 0$ such that $b = aq$.
$\#x$	Reads “the number of x”, which is the number of object x in the context.

List of Variables and Constants

n	The number of key-value pairs.
S	The set of keys. The i-th key is denoted as $S[i]$, $0 \leqslant i \leqslant n-1$.
V	The set of values. $S[i]$ corresponds to $V[i]$, $0 \leqslant i \leqslant n-1$.

a	The bucket size, i.e. the number of slots for values in a bucket.
b	The ratio of the number of elements to that of buckets. Given a fixed b, the number of buckets is then defined as $\lfloor \frac{n}{b} \rfloor$.
m	The number of buckets (in a level), defined as $m := \lfloor \frac{n}{b} \rfloor$.
k	The number of signature bits.
d	The upper bound of the signatures, defined as 2^k. All signatures fall into $[0..d-1]$.
r	The number of value bits per element.
h	The hash function (of a level) used to assign a bucket id to each key. The range will be $[0..m-1]$.
g	The signature function (of a level) with range $[0..d-1]$.
W	The cache line size in bytes.
t	The number of non-PHF levels in the data structure.
c	The number of possible cache misses per query at the last level (PHF level).

Glossary

ADT	Abstract data type.
BBST	Balanced binary search tree. Well-known data structure used to implement dynamic dictionaries.
CEFS	Cache-Efficient Function Store. The main contribution of the present thesis.
PHF	Perfect hash function. They are hash functions that do not cause any collisions..
BPZ Algorithm	A family of perfect hash functions proposed in [4].
CHD Algorithm	A family of perfect hash functions proposed in [5].
CHM Algorithm	A family of perfect hash functions proposed in [6]. In the present thesis the algorithm is used as a function store.
CMPH	C Minimal Perfect Hashing Library. A library to generate and to work with very efficient minimal perfect hash functions [7].
Intel PCM	Intel® Performance Counter Monitor [8], a set of programs to measure the CPU utilization of applications.
OpenMP	Open Multi-Processing, an API supporting multi-platform shared-memory parallel programming in C/C++ and Fortran.

1 Introduction

1.1 The Problem

Dictionaries are a kind of abstract data type (ADT) which handle key-value mappings. They support the following operations:

- Inserting a new key-value mapping into the dictionary;
- Removing a mapping from the dictionary. A mapping is specified by its key. The caller should be notified if no mapping for that key exists;
- Retrieving the value for a given key. The caller should be notified if no mapping for that key exists.

Dictionaries count as one of the most-used data structures. For example, an address book is a dictionary: the keys are names of the contacts, the value for a key consists of the phone number, the e-mail address, etc. of the contact. The aforementioned operations correspond to adding or removing a contact, or finding out the phone number or e-mail address of a contact. Balanced binary search trees (BBSTs) and hash tables are among the most well-known and most used implementations of dictionaries in the practice [9].

There exist many practical applications in which insertions and deletions never occur or are very rare, i.e. the key-value pairs are all given in advance. Examples of such applications include real dictionaries, which are usually revised at most once or twice a year. In such cases, the only operation that needs to be efficiently supported is the retrieval of values. We call such an ADT *static dictionary.* The original dictionaries supporting insertions and deletions are called *dynamic* accordingly. In the static case there are simpler algorithms and data structures. For example, the key-value pairs can be sorted by the key and stored in an array, allowing to answer queries with binary search. In this case, BBSTs are not that attractive any more because they are usually outperformed by binary searches due to large constant factors hidden in the big-oh notation [9].

Yet in many applications, the ability to detect invalid queries can also be dropped, e.g. when it is known in advance that the key being searched for is always in the data structure and we just want to get its value. An example would be the real-world dictionaries mentioned above. Dictionaries are made to be as extensive as possible. So it is to expect that every word we look for will be found. Another example of such applications are indices for reverse look-ups in database systems, that is, to find the row number of a given value in a table of unique values. In such applications, the queried values are often known to exist in the table.

The last scenario is the central problem we would like to tackle in the present thesis. The data structure in question can be formalized as the following definition.

Definition 1.1 (Function store). *Given a set of n unique keys $S[i]$ from a universe U, each having a corresponding r-bit value $V[i]$, $i \in [0..n-1]$, a* function store *is an abstract data type which supports the operation of retrieving the corresponding value for a key from S. The return value for a key not from S is unspecified*[1]*. In other words, a function store represents a static dictionary without the ability to detect invalid queries.*

A function store is also known as a *static support lookup table* in the literature [10]. The problem of implementing a function store was known as the *Retrieval Problem* [11]. The name "function

[1]"Unspecified" means the function either reports that the key does not belong to the set or it returns any valid value, though that value may not correspond to the key.

store" has been chosen because it sounds less technical without being ambiguous.

Note that the definition of function store implies that all keys and values are given beforehand, and insertion and deletion need not be supported, just like with static dictionaries. The connection between dictionaries, static dictionaries and function stores is the following:

$$\text{Dictionaries} \xrightarrow[\text{All data are given beforehand}]{\text{No update of any kind}} \text{Static Dictionaries} \xrightarrow{\text{Only valid queries}} \text{Function Stores.}$$

The most natural criteria for a good function store are short evaluation times and small space consumption. We therefore seek for an implementation of function store that is both fast and compact in terms of space usage.

1.2 Known Methods

Because function stores are constrained dictionaries, all implementation of dictionaries are automatically also function stores. However, since function stores give up much flexibility, we expect better performance from tailored implementations.

BBSTs and binary searches both answer queries within $\mathcal{O}(\log n)$ *comparisons*, which is too costly for large n and long keys. Naive hash tables use more than n buckets only to achieve *expected* constant time queries with no worst-case guarantee [9]. They all have to store the keys along with the values, which could take much space when the keys are long.

There are variants of hash tables that offer worst-case constant time queries. One of the most popular ones is the *Cuckoo Hashing* proposed in [12] and its variants, *d-ary Cuckoo Hashing* [13] and *Blocked Cuckoo Hashing* [14]. The key idea is to use an array of buckets, each having some constant number of slots ("blocked"), and each key can be in any of d possible buckets ("d-ary") instead of just one, calculated by d hash functions. In that way every query can be answered by checking at most d buckets, offering constant time guarantee. However, they also have to store the keys with them, and in the worst case they have to inspect d possibly not very small buckets.

Another family of competitive solutions are perfect hash functions (PHFs). A hash function f is said to be *perfect* with respect to a key set S if f maps keys in S to distinct integers in $[0..m-1]$ for some $m \geqslant n$. A perfect hash function is *minimal* if $m = n$. Perfect hash functions have been an active research area in the last two decades and many methods to generate them efficiently have been developed. The current state-of-the-art known to the author are the *CHD Algorithm* [5] and the *BPZ Algorithm* [4]. They can generate minimal perfect hash functions in expected $\mathcal{O}(n)$ time, and only $\Theta(n)$ bits are needed to describe such a function. Each function evaluation can be done in worst-case constant time. One of the best things about such perfect hash functions is that they do not have to store keys themselves. For keys not from S, they usually return an arbitrary integer.

Perfect hash functions offer a simple implementation of function stores: just generate a perfect hash function and use the function values as the array index into an array of all values. Such an implementation is fast enough for many applications, but not for all if the description of the PHF does not fit into cache. The evaluation using the BPZ Algorithm induces three or four possible cache misses depending on an internal parameter, because the algorithm inspects two or three locations in a big array to calculate an index, and then extracts the value from the value array at the calculated index. The CHD Algorithm also needs no less than two cache misses.

Yet another family of function stores exist. The basic idea is to assign k positions of an m-element array with r-bit elements to each key, and the value of the key is calculated as the bit-wise exclusive-or of those k elements. Therefore no access to another value array is needed, saving one cache miss per query. The idea has already been utilized in [6] two decades ago to construct perfect hash functions, though it has not been explicitly mentioned that the same idea can be exploited to implement function stores rather than just perfect hash functions. In the recent years, the same idea has been utilized in various ways in e.g. [11, 10] for function stores. we call this family of function stores the *CHM Algorithm*. They offer evaluation within k cache misses. $k = 2$ and $k = 3$ are typical values.

The focus of all those known methods is space-efficiency. Their space usage is known to be optimal or almost optimal (see e.g. [11]). The goal of the present thesis is to reduce the number of possible cache misses for the evaluation without sacrificing the space-efficiency too much.

1.3 Contribution of the Thesis

The main contribution of the present thesis is a new family of function stores, the Cache-Efficient Function Store (CEFSs), that offer very fast evaluations while being very compact in terms of memory consumption. The aforementioned perfect hash functions serve as an important component of the new function stores.

More precisely, the data structure can be built in expected $\mathcal{O}(n)$ time and supports worst-case constant time evaluation. The expected number of cache misses per query can be reduced to well under two under certain realistic conditions stated in Section 3.3. Besides the required memory for storing all values, only $\mathcal{O}(n)$ bits extra space is needed. The keys themselves are *not* stored.

The data structure is especially suitable for small values, e.g. machine-size words or smaller, in which case the number of cache misses of under two can be reached. For large variable-length values, the function store (with machine-size values) can be used as a perfect hash function, which can in turn implement general function stores, just like for other perfect hash functions described in Section 1.2. In this case, the CEFS is also competitive, because the number of cache misses introduced by fetching the value cannot be avoided in general. The nearest competitor with respect to evaluation times is the CHM Algorithm with $k = 2$ which, however, has a high space overhead.

The data structure is easily configurable. The performance for different parameter settings can be precisely predicted so that choosing the best parameters suitable to the application is no hard work. It is even possible to (at least semi-)automate the parameter selection.

1.4 Outline of the Thesis

Section 2 introduces the new method, which we call the Cache-Efficient Function Store (CEFSs). Section 3 discusses some details in the implementation of CEFS. The building procedure and the query operation will be explained in detail there. The mathematical analysis of CEFS follows in Section 4.

The new data structure is benchmarked and compared to the known methods in Section 5.

Section 6 concludes the thesis and discusses some possible extensions.

2 The New Approach

In this section, the new family of function stores, namely the Cache-Efficient Function Stores (CEFS), will be presented. The first subsection describes the structure of CEFS, and the following ones explain how to retrieve values from a CEFS and how the structure itself is built. Note that this section gives a high-level overview of the data structure and the algorithms. Some implementation details are given in Section 3.

2.1 The Cache-Efficient Function Stores (CEFS)

A CEFS is a conceptually recursive data structure consisting of multiple levels. Figure 1 gives an overview of the structure.

A CEFS contains $t + 1$ levels, with the last level being any perfect-hash-function-based function store, which serves as a "fallback". Each level except for the last one consists of an array of m buckets. Every bucket can hold up to a values and contains the encoded *signatures* of the keys whose values are stored in the bucket. As will see shortly, different levels normally have different and decreasing m's.

The rationale for the introduction of such buckets is that we hope to fit each bucket into a cache line. As we will see in Section 4, the expected number of bucket accesses for each query can be bounded well under two, therefore each query can be answered with expected less than two cache misses.

Signatures are introduced to eliminate the need to store the keys themselves. They serve to identify keys in each bucket. Since the keys are not stored as mentioned earlier, some sort of information on the keys must be recorded somewhere or exploited somehow in order to select the correct value when searching for a key. The encoded signature holds that piece of information. Since there is e than one slot in the bucket, the encoding also contains information of which slot each key belongs to. Details are given in Section 3.2.

Every level also holds two hash functions, which are chosen independently of the ones in other levels. One, called h, is used to distribute the keys into buckets and the other (by assigning an integer from $[0..m-1]$ to every key), g, generates k-bit-long signatures for the keys.

The following parameters are needed to build a CEFS:

- a: positive integer, the number of slots in every bucket.
- b: positive number, used to determine the number of buckets m in a level. m is defined by $m := \left\lfloor \frac{n}{b} \right\rfloor$ where n is the number of elements for the current level at the beginning.
- k: positive integer, the number of signature bits for each key. We also define $d := 2^k$. Thus d is the upper bound of all signatures and $g(x)$ returns an integer in $[0..d-1]$.

More precisely, for a sequence of n keys $S[0..n-1]$, the first level of CEFS has $m := \left\lfloor \frac{n}{b} \right\rfloor$ buckets[2], numbered 0 to $m-1$, each of which contains slots for up to a values, as well as the encoded signatures of their corresponding keys. Hash function h maps every key to an integral bucket id in $[0..m-1]$, g generates a k-bit signature for each key. If a key x is stored in this level, then it must be in bucket number $h(x)$. It may happen, however, that some keys cannot be stored

[2]For the sake of conciseness we may leave out floor/ceiling/rounding operators on fractions as long as they do not affect the asymptotic behavior of the data structure. The default behavior in this thesis is flooring.

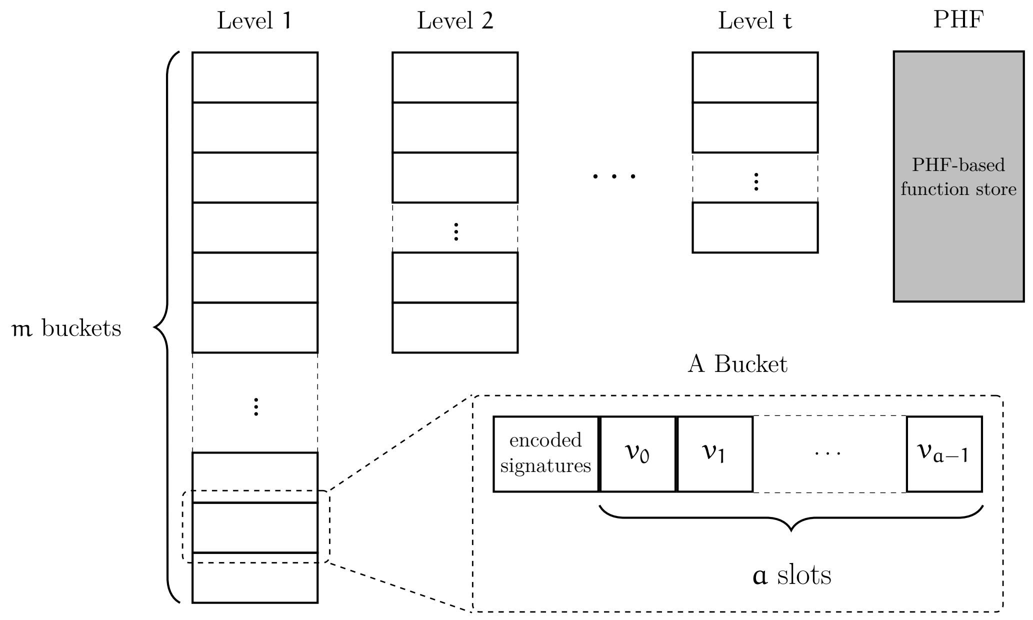

Figure 1: Schematic diagram of a CEFS. It contains $t + 1$ levels, with the last level being any perfect-hash-function-based function store. Each level except for the last one consists of an array of m buckets. Each bucket can hold up to a values and contains the encoded signatures of the keys whose values are stored in the bucket. Note that different levels normally have different and decreasing m's.

in the first level due to signature collision or capacity limitation in their buckets. Those keys therefore *fall* into the next level, which has the same shape but different number of buckets, except for the last level which is a PHF-based function store.

To make a clear image of how a CEFS works, we first take a look at how queries are done. The building procedure will be described thereafter.

2.2 Retrieving a Value from a CEFS

The evaluation of a key, i.e. retrieval of its value, is quite simple on a CEFS. Assume we want to find the value for a key $x \in S$. We search each level in order, with the following procedure.

In each (non-PHF) level, we first calculate $h(x)$, which is the bucket id of x if it was stored in this level. Then we check whether its signature $g(x)$ appears in the (encoded) signatures of that bucket. If it does, we simply return the value stored in the corresponding slot in the bucket, according to the positional information encoded in the signatures. Otherwise we conclude that the key was not stored in this level. We proceed to the next one.

The value of the key will eventually be found if it was stored in the CEFS. For keys that were not stored in the CEFS, the algorithm above would generally return an arbitrary value. Invalid searches may only be detected in rare cases where the perfect hash function in the last level supports that to some extent, and the signature for that key does not appear in any buckets it may belong to in all non-PHF levels.

It is worth emphasizing that only one bucket is accessed in every non-PHF level. As we will see in Section 3.3, the number of cache misses per query can be characterized by the number of bucket accesses for many parameter combinations.

Algorithm 1 gives an overview of the evaluation process.

Algorithm 1: The Evaluation Algorithm

```
Input: CEFS $f$ and $key \in U$
Output: The corresponding value of $key$ if $key \in S$; otherwise unspecified.
begin
    for $i = 1$ to $t$ do                                          // For each level i...
        bid ← $h_i(key)$                                           // The bucket id
        sig ← $g_i(key)$                                           // The signature
        if sig is in Bucket bid then
            return the corresponding value in the bucket
        end
    end
    if f has a PHF-level phffs then
        val ← the corresponding value of key in phffs              // Can be arbitrary!
        return val
    else
        return Not found!
    end
end
```

2.3 Building a CEFS

We first choose parameters for the CEFS according to the memory constraints, timing goal, etc. Section 3 contains more details on the choice of parameters.

As the evaluation procedure indicates, we build a CEFS level by level. For each level, we select (independently of other levels) two hash functions h and g for generating bucket ids and signatures. All keys are divided into the m buckets according to the h values. For each bucket we have to ensure that two requirements are met:

(i) No two elements in the bucket have the same signature. This makes sure that the evaluation algorithm works.
(ii) The capacity of the bucket is not exceeded, i.e. no more than a elements are in the bucket.

Therefore we do the following, in that order, for each bucket:

Collision Resolution. If two or more elements share the same signature, none of them will be stored in the current level. We remove all of them from the bucket and mark them for the next levels.

Capacity Compliance. If there are more than a elements remaining in the bucket, choose any a of them for the current level, and mark the others for the next levels.

Note the above order is important to ensure the evaluation algorithm works correctly.

After these steps, we generate the signature encoding for the keys that are not marked in each bucket and store the encoding in that bucket. Values of the keys that stay in the buckets are also stored in the bucket, positioned according to the encoding.

If there are elements marked for the next level, we continue to build that level. Otherwise we are done. If, however, we reach a certain depth or the number of remaining elements is under a certain threshold, we build a PHF-based function store for those elements instead of building the next level as normal.

The building algorithm is given as Algorithm 2. Note the sorting procedure involved in the pseudo-code can be done in linear time since bucket ids are integers in $[0..m-1]$.

Algorithm 2: The Building Algorithm

```
Input: $S[0..n-1]$, $V[0..n-1]$ and parameters $(b, k, a, t)$
Output: A CEFS $f$
begin
    $lv \leftarrow 0$                                        // The current number of levels
    $curS \leftarrow S$, $curV \leftarrow V$, $curN \leftarrow n$
    while $curN > 0$ do                                      // There are unsettled elements
        $lv \leftarrow lv + 1$
        if $lv > t$ or $curN < threshold$ then               // Reached threshold for PHF-level
            $phffs \leftarrow$ PHF-based function store for $(curS, curV)$
            Store $phffs$ in $f$
            return $f$
        else                                                 // Building next CEFS level
            Choose two random hash functions $h_{lv}$ and $g_{lv}$
            Calculate the bucket id $h_{lv}(key)$ for all $key \in curS$
            Sort all keys according to $h_{lv}(key)$         // Distributing keys into buckets
            foreach Bucket $A_i$, $i \in [0..m-1]$ do
                Calculate the signatures $g_{lv}(key)$, $key \in A_i$
                if signature $s$ occurs more than once then  // Collision Resolution
                    foreach $x \in A_i$ with $g_{lv}(x) = s$ do
                        Mark $x$ for the next level
                    end
                end
                if $|A_i| > a$ then                          // Capacity Compliance
                    Choose $a$ keys and mark others for the next level
                end
                Store the signatures and values of the unmarked keys in this level.
            end
            $curS \leftarrow$ the marked keys
            $curN \leftarrow |curS|$
            Set $curV$ accordingly
        end
    end
end
```

3 Implementation Details

In this section we discuss some details and options in implementing the CEFS.

3.1 Choosing Hash Functions

The CEFS uses hash functions internally. We assume in the theoretical analysis in Section 4 that these are fully random. In practice, however, it appears to be enough to use a fast deterministic heuristic hash function to achieve sufficiently good performance.

In the experiments, the author used a family of fast non-cryptographic hash functions proposed by Jenkins [15], which we call the *Jenkins Hash*. It proves to be very suitable for CEFS in that every hash function in the family needs only 4 bytes (which is the seed) to be represented and the hash values are sufficiently well distributed [5, 16]. The former property makes the "metadata" (consisting of information on every level, such as the number of buckets(m) in the level, the hash functions for the level, pointer to the array of buckets of the level) as small as possible, which in turn contributes to the cache-efficiency. The latter ensures that few or no buckets are so full that too many elements fall into next levels.

With hash functions like the Jenkins Hash, we can simply choose some pseudo-random number as the seed in each level in order to get a new hash function from the family that is pseudo-independent of the ones in the former levels.

The full-randomness assumption can be justified with a "split-and-share" technique introduced in [17], which can be applied to generate fully random hash functions on S with probability $1 - n^{-c}$ for some $c > 0$, at the cost of $o(n)$ extra space.

Another problem concerning the hash functions is that we need two hash functions for each level according to the description of the CEFS. But given the fact that many hash function return sufficiently long values (e.g. 32, 64 or even 128 bits for the Jenkins Hash), it is unnecessary to have two distinct hash functions for our purpose. We can just use a single hash function in each level, using the higher and lower bits of the hash values for the signatures and bucket ids, respectively. This way we make the metadata even smaller and save one hash function evaluation per level for each query while having to pay little penalty in the quality of the generated signatures and bucket ids .

3.2 Signature Encoding

For the sake of comprehensibility the details about the signature encoding were left out from the description of CEFS. The signature encoding is in essence an abstract data type supporting the following operations:

$E = \text{build}(sigs)$ which builds the encoding E for a list $sigs$ of at most a signatures. The caller has to ensure a signature is not present twice.

$pos = \text{find}(E, sig)$ which tries to find the array index for signature sig in the encoded signatures E. If sig does not appear in E, the caller should be notified (e.g. through some sentinel return value like -1).

Two simple and practical encoding schemes are presented in this section. Brief introduction to an optimal scheme (assuming every bucket is independently considered) is also given. Details are given in Appendix A.

3.2.1 Bit-Vector Encoding

In a bit-vector encoding, the list of (signature, value) pairs is assumed to be *sorted* by signature. The (at most) a signatures in the bucket are represented by a bit vector of length $d := 2^k$ with at most a 1's in it. Values are stored in the first slots of the buckets. The two operations are implemented as following:

build($sigs$) returns a bit vector $E[0..d-1]$ of length $d = 2^k$. $E[s] = 1$ iff a signature s is present, otherwise $E[s] = 0$.

find(E, sig) returns the number of ones in $E[0..sig-1]$ if $E[sig] = 1$, otherwise it returns -1. Note this is correct only when the (signature, value) pairs was sorted by the time of creation of E.

The total memory requirement of the encoding for a bucket is $d = 2^k$ bits, independent of a. For small d, e.g. $d = 64$ ($k = 6$), all signatures of a bucket can be encoded in a 64-bit integer. This allows compact signature representation and fast find-operations by utilizing *popcnt*-instructions (which counts the number of ones in a binary number) provided by many CPUs [18, 19].

3.2.2 Fixed-Length Encoding

Another equally simple encoding scheme is the fixed-length encoding. Each signature in the bucket is represented by exactly k bits, and they are concatenated to form the encoded signatures with $a \cdot k$ bits. Note that a bucket can also hold less than a elements, which means we have to be able to identify whether a slot contains a value for some key or is just garbage. We have at least three ways to handle non-full buckets:

(i) Store the actual number of elements in the bucket. This needs about $\log_2 a$ bits.
(ii) Assign some bucket-local sentinel signature for all the unoccupied value slots. This sentinel signature is generated *before* the collision resolution, so that the sentinel will not be taken as the signature of any actual element that could potentially be stored in the bucket. The problem here is that it is not always possible to find such a sentinel for each bucket due to overfull buckets.
(iii) We can also reserve a signature value, e.g. 0, as a global sentinel for the whole CEFS, so that every valid signature is from $[1..d-1]$ instead of $[0..d-1]$. This does not impose much penalty in practice, but needs some minor modifications in the analysis.

The building procedure is straightforward (except for finding a sentinel if the second approach is used). Queries can be done by running through all signatures in the bucket and see whether the signature being searched for is present and, if it is, in which position.

3.2.3 An Optimal Encoding Scheme

Assuming every bucket is independently processed, i.e. the list of all signatures in the level is not considered as a whole (in which case the probabilistic distribution of possible signature lists must be taken into account), it is possible to construct an "optimal" encoding scheme in the sense that every possible signature list has a unique encoding, and every encoding represents a signature list. That is to say, there exists a bijection between possible encodings and lists of signatures. A possible such bijection is given in Appendix A.

The problem is that the decoding procedure takes much more, though constant, time than other two encoding schemes. It is therefore not preferred for query-intensive applications.

3.3 Cache Awareness

Each evaluation in a CEFS accesses some buckets. The number of bucket accesses is bounded to a constant, i.e. the depth of the whole data structure. It is shown in Section 4.2 that the expected number of bucket accesses is even well under two for many parameter settings.

Under certain realistic assumptions, the phrase "cache misses" can be used as a synonym of "bucket accesses":

Assumption 1. Every bucket is small enough to be held in one cache line. Since each bucket holds a values, values should not be too large. For large values, CEFS can be used as a perfect hash function: it maps each key to the index of its value in the array of all values. In this case one or multiple additional cache misses will be induced only to fetch the value from memory, which is by intuition unavoidable. This assumption is true for many parameter combinations, CPU architectures and scenarios, e.g. using bit-vector signature encoding explained in Section 3.2, $(a, k, r) = (4, 4, 32)$ yields buckets of size $2^k + a \cdot r = 2^4 + 4 * 32 = 144\,\text{bit} = 18\,\text{byte}$ which is much smaller than a currently typical cache line size of 64 bytes [18, 20].

Assumption 2. Buckets are properly (cache-)aligned. This can be achieved easily in many relatively low-level languages like C/C++ (see e.g. [21]). Note this could result in waste of memory. Parameters should be chosen so that the cache line size is equal to or slightly bigger than a multiple of the size of a bucket to make full use of each cache line.

Assumption 3. Metadata of the levels do not contribute to the number of cache misses. Metadata are small for each level. 16 bytes is a typical size: 8 bytes for the pointer to the bucket array; 4 bytes for the hash function of the level and 4 bytes for m for calculating bucket ids. Since the number of levels is bounded to a small constant (e.g. 8), metadata of the whole data structure are also of constant size and thus do not impose much penalty on the number of cache misses. The same assumption is also made when inspecting the cache behavior of previously known methods, therefore this does not damage the fairness of the comparison.

3.4 Choosing Parameters

There are at least three parameters to set in order to build a CEFS as stated in Section 2.3:

a: the capacity of each bucket.
b: used to determine the number of buckets m in a level ($m := \lfloor \frac{n}{b} \rfloor$).
k: the number of signature bits.

These parameters determine the shape and thus the performance of a CEFS. Choosing them wisely is crucial to fast CEFSs.

With the deduced results in Section 4, it is even possible to predict the final space consumption quite precisely. Given the fact that the parameter space is not too large (except that b does not have to be integral which is not that important since we can always choose the nearest integer), we could just run through all combinations, filtering out the unacceptable ones due to cache, time or space constraints and select the best one for our application according to certain metrics or goals. This allows us to easily select the best parameters fitting a certain application and/or performance goal, e.g. it is possible to select the setting "requiring the least number of cache misses per query while constraining the space overhead to under 2 bytes".

From this point of view, the CEFS is actually a quite flexible and "adaptive" data structure in that the best parameters can actually be chosen automatically.

3.5 Parallelization

The evaluation algorithm does not need to be parallelized, because the number of levels is usually a small constant (e.g. 8).

On the other hand, it is foreseeable that the CPU-intensive building procedure may take some time for exceedingly large sets, even though the algorithm is already very fast. Thus it makes sense to parallelize the building procedure to achieve better CPU utilization and performance.

Fortunately the bottlenecks of the building algorithm can be parallelized:

Distributing Keys into Buckets. One way to implement this is to simply sort the keys by their bucket id. In this way all elements in the same bucket will be together. Parallel sorting is a well researched area, many algorithms exist, and some compilers even offer a very simple switch to parallel algorithms, see e.g. [22].

Collision Resolution and Capacity Compliance. This is actually done for each bucket independently of other buckets and thus naturally parallelizable.

Having parallelized the bottlenecks, the building algorithm runs very fast, faster than any known methods by magnitudes, see experimental results in Section 5.

4 Analysis

We have already seen the conceptual simplicity of the CEFS. It is natural to ask about its performance. In this section we analyze the data structure carefully and thereby show the performance guarantee offered by such a conceptually simple data structure.

We make the following assumptions for the analysis to work:

(i) The underlying hash functions distribute all keys uniformly at random. As such, each key can be assigned to any of the m buckets with probability $\frac{1}{m}$ and each key can get any of the 2^k signatures with probability $\frac{1}{2^k}$.
(ii) n is sufficiently large, at least $n > a$. This is merely a simplifying assumption unless noted otherwise.

4.1 Falling Proportion

In order to analyze the performance of the whole data structure, it is helpful to be able to calculate the expected number of elements that do *not* fit in the current level (thus falling into next levels due to collision with other elements or saturation of the assigned bucket). We define the expected proportion of these elements to be $p(n, m, a, d)$ and call it the *falling proportion.*

Let $N_i(n, m, a, d)$[3] be the number of elements assigned to Bucket i, and $n_i(n, m, a, d)$ be the number of those staying there in the end. The falling proportion being sought for is then $\frac{n - E\left[\sum_{i=0}^{m-1} n_i\right]}{n} = 1 - \frac{E\left[\sum_{i=0}^{m-1} n_i\right]}{n}$. Because of the linearity of expectation and the uniform randomness of the hash functions, this can be further reduced:

$$p(n, m, a, d) = 1 - \frac{E\left[\sum_{i=0}^{m-1} n_i\right]}{n} = 1 - \frac{\sum_{i=0}^{m-1} E\left[n_i\right]}{n} = 1 - \frac{m \cdot E\left[n_t\right]}{n} = 1 - \frac{E\left[n_t\right]}{b}$$

where t can be any fixed integer in $[0..m-1]$ in the last expression. Expanding $E\left[n_t\right]$ yields

$$\begin{aligned} E\left[n_t\right] &= \sum_{s=1}^{n} s \cdot \Pr\left[n_t = s\right] && \text{(Definition of expectation)} \\ &= \sum_{s=1}^{n} s \sum_{r=1}^{n} \Pr\left[N_t = r\right] \cdot \Pr\left[n_t = s \mid N_t = r\right] && \text{(Law of Total Probability)} \\ &= \sum_{r=1}^{n} \sum_{s=1}^{n} s \cdot \Pr\left[N_t = r\right] \cdot \Pr\left[n_t = s \mid N_t = r\right] && \text{(Swapping sums)} \end{aligned}$$

The last expression is a more natural way of thinking: we first distribute r keys into the bucket in question (with probability $\Pr\left[N_t = r\right]$), and then calculate the probability that exactly s of them stay there ($\Pr\left[n_t = s \mid N_t = r\right]$). It remains to calculate the two probabilities involved in the expression.

[3] Parameters are left out when it is clear from the context what values they take. Furthermore, in a function, constants and variables which are merely simple combinations of the parameters could be used. For example, b could appear in a function which takes parameters n and m.

4.1.1 $\Pr[N_t = r]$

Given n elements, each assigned to one of m buckets uniformly at random, we are to calculate the probability that a specific bucket t has exactly r elements in it.

This problem is quite easy because $N_t \sim B(n, \frac{1}{m})$, since every element can either be in Bucket t (with probability $\frac{1}{m}$) or not be there (with probability $1 - \frac{1}{m}$). This results in $p1(n, m, r)$

$$p1(n, m, r) := \Pr[N_t = r] := \binom{n}{r} \left(\frac{1}{m}\right)^r \left(1 - \frac{1}{m}\right)^{n-r}$$

4.1.2 $\Pr[n_t = s \mid N_t = r]$

After distributing elements into buckets, we take care of collisions and the capacity concerns as described in Section 2.3.

The problem here can be formulated as following: Given r elements in a bucket (of size a) and d possible signatures, we would like to find the probability that exactly s ($s \leqslant a$) of them stay there in the end after considering collisions and the capacity of the bucket.

The problem is naturally divided into two parts according to whether the capacity of the bucket is exceeded after resolving collisions (by assigning all colliding elements into next levels), where exceeding the capacity results in $n_t = a$. Therefore we distinguish the cases $s < a$ and $s = a$.

4.1.2.1 $s < a$. We first tackle the case $s < a$ where no second step is performed. The probability needed is simply the number of ways to assign signatures to elements so that exactly s occupy their signatures exclusively divided by the number of all possible assignments. Since the latter is obviously d^r, the only problem is to calculate the former.

We divide the d signatures into three (possibly empty) parts:

(i) s signatures which are assigned to exactly one element
(ii) u unoccupied signatures
(iii) $d - s - u$ signatures which are all assigned to two or more elements.

The calculation can therefore be naturally divided into three stages.

For the first stage, there are clearly $\binom{d}{s}$ possible choices of those s signatures. For every possible selection of signatures, we assign them to a subset of s elements. For every choice of s signatures and s elements, there are $s!$ possible concrete assignment. The multiplication principle of combinatorics results in a total of $\binom{d}{s}\binom{r}{s}s!$ possibilities for the first stage.

The second stage is easy, just choose u from the remaining $d - s$ signatures. The possibilities are counted by $\binom{d-s}{u}$.

The last stage is a bit trickier: we would like to assign the remaining $d - s - u$ signatures to $r - s$ elements such that every signature is used no less than twice. One could make use of the so-called *2-associated Stirling numbers of the second kind* where $\left\{ {n \atop k} \right\}_2$ counts the ways to partition a set of n labeled objects into k unlabeled boxes such that every box contains at least two objects. There is (to the best of the author's knowledge) unfortunately no known closed formula to calculate $\left\{ {n \atop k} \right\}_2$ (although an explicit formula does exist, see e.g. [2, 3]), but they satisfy the following recurrence relationship [1, 2, 3]:

$$\left\{ {n \atop k} \right\}_2 = \begin{cases} 0 & \text{if } n < 2k \text{ or } n < 0 \text{ or } k < 0 \\ 1 & \text{if } n = k = 0 \\ k\left\{ {n-1 \atop k} \right\}_2 + (n-1)\left\{ {n-2 \atop k-1} \right\}_2 & \text{else} \end{cases}$$

With help of those numbers, regarding the signatures as the boxes and the elements as the objects, there are $\left\{ {r-s \atop d-s-u} \right\}_2$ possibilities. Note that the 2-associated Stirling numbers distribute objects into *unlabeled* boxes, which means we need to multiply the result by $(d-s-u)!$ to take every permutation of the signatures into account. The total count for this stage is then $\left\{ {r-s \atop d-s-u} \right\}_2 (d-s-u)!$.

Relating all three stages with the multiplication principle and summing up the result for every u, the final result *for the case* $s < a$ is calculated by the function $f(d, r, s)$:

$$f(d, r, s) := \frac{\sum_{u=0}^{d-s} \overbrace{\binom{d}{s}\binom{r}{s} s! \cdot}^{\text{Stage 1}} \overbrace{\binom{d-s}{u} \cdot}^{\text{Stage 2}} \overbrace{\left\{ {r-s \atop d-s-u} \right\}_2 (d-s-u)!}^{\text{Stage 3}}}{d^r}$$

4.1.2.2 $s = a$. The formula above only grasps a portion of the truth for the case $s = a$, since $s = a$ could also possibly be resulted from exceeding the capacity after collision resolution. This case is complementary to the one above, which means the probability for $s = a$ is $g(d, r, s)$:

$$g(d, r, s) := \overbrace{f(d, r, a)}^{\text{Capacity reached}} + \overbrace{\left(1 - \sum_{i=0}^{a} f(d, r, i)\right)}^{\text{Capacity exceeded}}$$

Combining both cases $s < a$ and $s = a$ gives the needed probability for the specific Bucket t $p2(d, r, s)$:

$$p2(d, r, s) := \Pr[n_t = s \mid N_t = r] = \begin{cases} f(d, r, s) & \text{if } s < a \\ g(d, r, s) & \text{if } s = a \end{cases}$$

And finally the expected value being sought for:

$$\begin{aligned} E[n_t] &= \sum_{r=1}^{n} \sum_{s=1}^{n} s \cdot \Pr[N_t = r] \cdot \Pr[n_t = s \mid N_t = r] \\ &= \sum_{r=1}^{n} \sum_{s=1}^{n} s \cdot p1(n, m, r) \cdot p2(d, r, s) \\ &= \sum_{r=1}^{n} p1(n, m, r) \sum_{s=1}^{n} s \cdot p2(d, r, s) \\ &= \sum_{r=1}^{n} \underbrace{\binom{n}{r}\left(\frac{1}{m}\right)^{r}\left(1 - \frac{1}{m}\right)^{n-r}}_{p1(n,m,r)} \cdot \underbrace{\left(\sum_{s=1}^{a} s \cdot f(d, r, s) + a\left(1 - \sum_{i=0}^{a} f(d, r, i)\right)\right)}_{\sum_{s=1}^{n} s \cdot p2(d,r,s)} \end{aligned} \tag{4.1}$$

With $E[n_t]$ at hand, the expected number of elements falling into the next levels (and thus also that of those staying in the current level) and its proportion $p(n, m, a, d)$ can be easily calculated.

At this point, i.e. without having a closed formula for $E[n_t]$ or for the falling proportion $p(n, m, a, d)$, we would like to make the following conjecture, which would greatly simplify further analysis of the performance:

Conjecture 4.1. *For fixed parameters (a, d, b) satisfying $b \mid n$, the falling proportion $p(n, \frac{n}{b}, a, d)$ converges against a real number $0 < p < 1$ as $n \to \infty$.*

This conjecture is supported by various evidences in the practice. For example, Figure 2 gives an overview of how the p values converge as n increases for two sets of fixed (b, k, a) parameters. Furthermore, comparing the measured proportion[4] of elements falling into next levels with $n = 10\,000\,000$ to the computed values with $n = 500$ for all combinations of $(b, k, a) \in [2..16] \times [2..6] \times [2..16]$, no difference larger than 0.017 has been observed. To gain even more confidence, we could show the following theorem and find more evidence in its proof:

Theorem 4.2. *For fixed parameters (a, d, b) satisfying $b \mid n$, the falling proportion $p(n, \frac{n}{b}, a, d)$ converges against a real number $p \in [0, 1]$ as $n \to \infty$.*

Proof. Since $p1(n, m, r)$ is actually the probability mass function of the binomial distribution $B(n, \frac{1}{m})$ and $n \cdot \frac{1}{m} = b$ is a constant, it is well known that $B(n, \frac{1}{m})$ can be approximated using the Poisson distribution $\text{Pois}(b)$ when n becomes large (see e.g. [23]), and thus

$$p1(n, m, r) \xrightarrow{n\to\infty} \frac{b^r}{r!} \cdot e^{-b}.$$

Note the last expression is merely a function in r instead of in n and r.

Consider the sum $S(n)$ achieved by replacing the $p1(n, m, r)$ part with $\frac{b^r}{r!} \cdot e^{-b}$ in the last sum in Equation 4.1. $S(n)$ is then the partial sum of a series with positive items and thus increasing. Furthermore, $S(n)$ is bounded by a, because the second part of the sum is by definition bounded by a and the first part is a probability mass function (which sums to 1 as $n \to \infty$). Thus $S(n)$ converges.

Since each term in $S(n)$ is the limit of the corresponding term in $E[n_t]$, their quotient converges against 1. Thus both series share the convergence according to a form of the Comparison Test (see e.g. [24]). Therefore $E[n_t]$ converges, so does $p(n, \frac{n}{b}, a, d)$. The latter as a proportion is naturally bounded in the interval $[0, 1]$. □

However, the proof did not show that the limit p is not exactly 1. But it brings some confidence that the limit is *near* that of $S(n)$. Because $S(n)$ is increasing, the falling proportion will be *decreasing* as n increases if we use the Poisson distribution instead of binomial distribution, which means we can achieve an upper bound on the falling proportion, which is also useful in practice. Figure 3 shows how the p values would look using $S(n)$ comparing to using $E[n_t]$ for calculating the falling proportion.

Using $0 < p < 1$ as a fixed parameter, many properties of the data structure can be deduced. These are summarized in the following sub-sections.

4.2 Time Properties

Fast queries are the most important design goal of this data structure. The resulting data structure is indeed fast: it allows worst-case constant time queries with minimal number of cache misses when backed up with the optimizations described in Section 3.

[4] Up to the first three levels to eliminate the effect of "small" levels on the statistics.

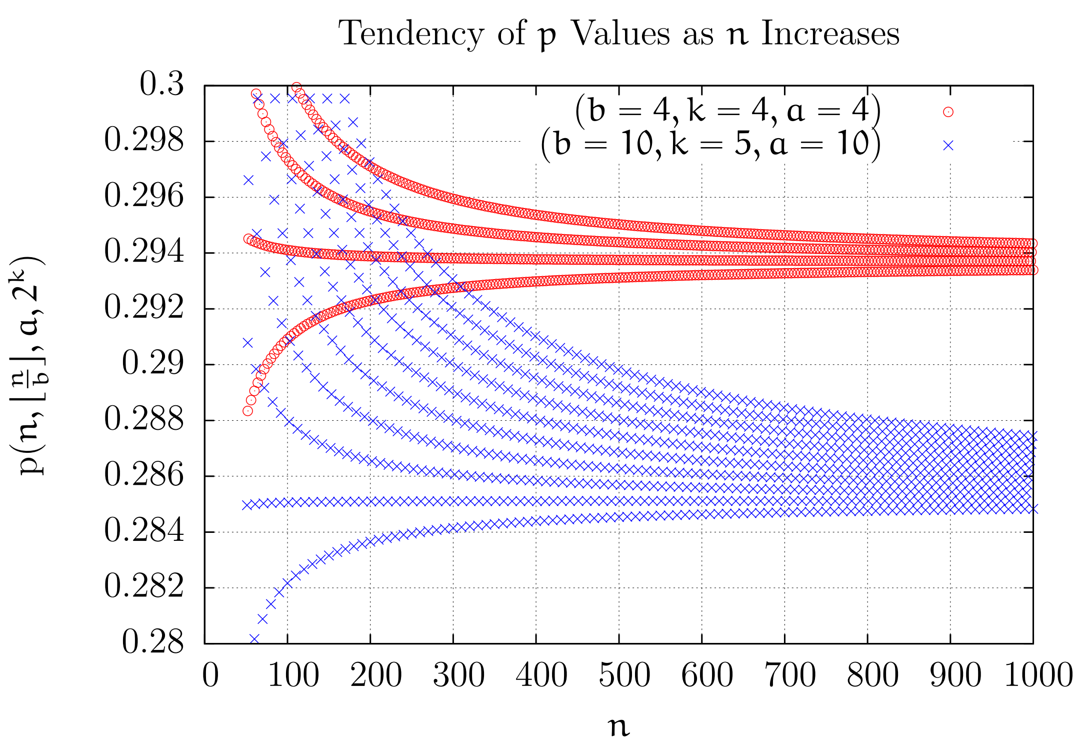

Figure 2: The p values for $n \in [50..1000]$, $m = \lfloor \frac{n}{b} \rfloor$ with fixed parameters $(b, k, a) = (4, 4, 4)$ and $(b, k, a) = (10, 5, 10)$. There are multiple lines for each configuration because the same number of buckets are used for $n \in [tb..tb + b - 1]$, $t = 0, 1, \dots$, with increasing p values. Therefore those are in fact many increasing runs of length b. The p values converge against about 0.294 and 0.286 respectively quite quickly as n increases.

4.2.1 Building a CEFS

The asymptotic time complexity of the building procedure is analyzed in this section. We show a CEFS can be built in expected $\mathcal{O}(n)$ time.

The building procedure in each level involves distributing keys into buckets and processing each bucket independently. Other parts of the building algorithm are obviously implementable in linear time.

The distribution can be done by simply sorting the keys by their bucket ids. Since each bucket id is within $[0..m - 1]$, the sorting can be done in linear time with a non-comparison-based sorting algorithm like Counting Sort. Keys in every bucket can also be sorted by their signatures by the way, forming a Radix Sort, in order to simplify the collision resolution and capacity compliance procedure.

For each bucket, the collision resolution and capacity compliance can be done in time proportional to the number of elements in that bucket just by running through all elements in the bucket once, keeping record of the last seen signature.

Since we have a constant number of non-PHF levels, the total time spent here is $\mathcal{O}(n)$ in the worst case. For the last PHF level, the expected construction time is linear when the BPZ Algorithm or the CHD Algorithm is used.

Therefore the overall time complexity of building a CEFS is expected $\mathcal{O}(n)$.

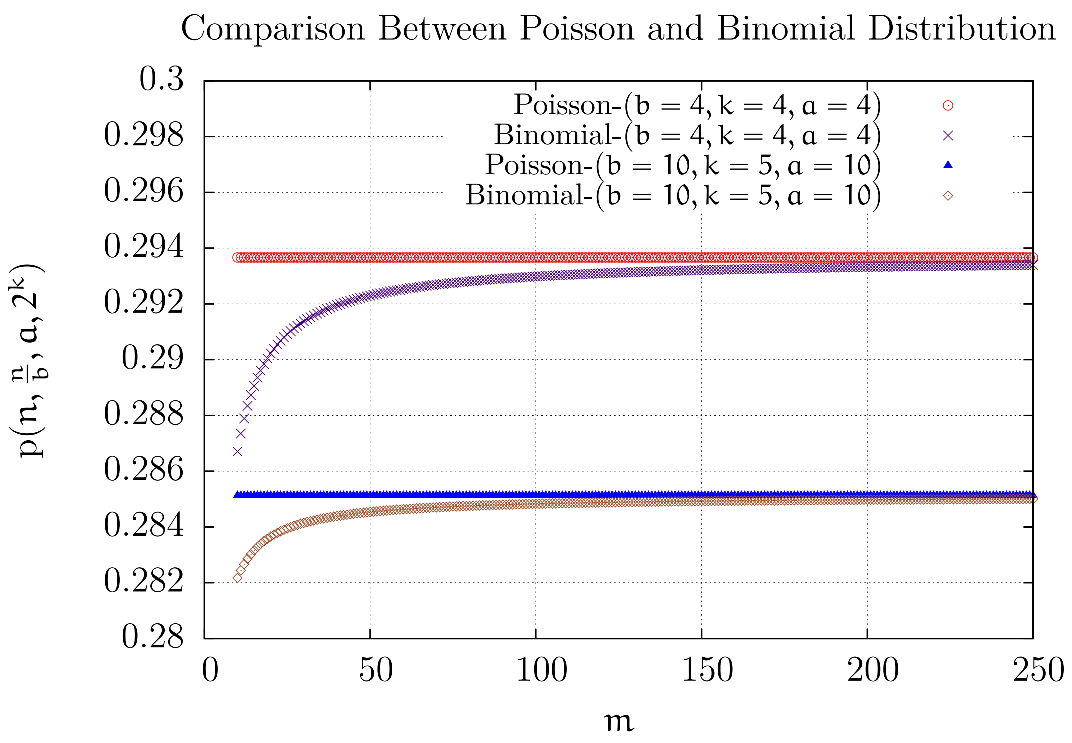

Figure 3: The p values for $m \in [10..250]$, $n = mb$ with fixed parameters $(b, k, a) = (4, 4, 4)$ and $(b, k, a) = (10, 5, 10)$. We see that the p values calculated using binomial distribution is less than using Poisson distribution, but they go together very quickly. The graph for the Poisson distribution even appears like a constant.

4.2.2 Querying a Key

Thanks to the perfect hash function in the last level, the query time is bounded to $\mathcal{O}(1)$. So we concentrate on a precise average-case analysis.

The expected depth of a query characterizes the expected number of bucket accesses and thus the number of cache misses, as justified in Section 3.3. This can be calculated with the definition of expected values:

$$\mathrm{E}[\#\text{cache\_misses}] = \mathrm{E}[\text{depth}(key)] = \sum_{i=1}^{\infty} i \cdot p^{i-1} \cdot (1-p) = \frac{1}{1-p}$$

where the last step results from some standard manipulations of series (see e.g. [24]), recalling that p is assumed to be within the open interval $(0, 1)$. Similar manipulations are also utilized later in this section.

The expression above, however, assumes that the data structure is used for every level and this is indeed beneficial over known methods[5] in theory if $p < \frac{1}{2}$, in which case $\mathrm{E}[\#\text{cache\_misses}] < 2$. If we utilize other function stores as fallback for Level $t + 1$, providing access with no more than c cache misses, the number of cache misses for a query would not exceed $t + c$ in the worst case. This setting guarantees constant worst-case query time both in theory and practice with

[5] Assuming two cache misses per query is achieved.

expectation (“cache misses – bounded”):

$$\begin{aligned} E[\#\text{cache\_misses}_b] &= \sum_{i=1}^{t} i \cdot p^{i-1} \cdot (1-p) + p^t \cdot (t+c) \\ &= \frac{1-(1-c(1-p))p^t}{1-p} \end{aligned}$$

4.3 Space Properties

This subsection shows that the data structure is compact as well as fast.

It is in fact quite easy to see that the data structure takes $\mathcal{O}(n)$ space in the worst case if backed up with perfect hash functions: we have constant number of levels, each needing $\mathcal{O}(n)$ space.

Besides the asymptotic complexity, we can even calculate the actual space usage on average in great precision.

4.3.1 Number of Buckets

Since every bucket has a constant size, knowing the number of buckets in total is helpful to estimate the space consumption of the whole data structure.

Assume n keys are given as input for a specific level[6]. There are $m = \frac{n}{b}$ buckets in this level. Noting that the number of input keys on the i-th level is np^{i-1}, summing over all levels results in:

$$E[\#\text{buckets}] = \sum_{i=1}^{\infty} \frac{np^{i-1}}{b} = \frac{n}{(1-p)b}$$

Note that this sum goes over all i. This holds under the assumption that all levels are non-PHF ones. If we bound the number of non-PHF levels to no more than t, the expected number of buckets is then

$$E[\#\text{buckets}_b] = \sum_{i=1}^{t} \frac{np^{i-1}}{b} = \frac{n}{(1-p)b}(1-p^t)$$

4.3.2 Total Size

Assume each stored value has r bits and a signature encoding scheme using sig_size(a, k) bits for a bucket of size a is used. Knowing that there are #buckets buckets in total and each bucket uses (sig_size$(a, k) + a \cdot r$) bits, the total size of the whole data structure can be obtained:

$$\begin{aligned} E[\text{tot\_size}] &= E[\#\text{buckets}] \cdot (\text{sig\_size}(a,k) + a \cdot r) \\ &= \frac{n}{(1-p)b} \cdot (\text{sig\_size}(a,k) + a \cdot r) + \mathcal{O}(1) \text{ bit} \end{aligned}$$

For the two simplest signature encoding schemes bit-vector encoding and fixed-length encoding described in detail in Section 3.2, the function sig_size(a, k) for a bucket is simply 2^k and $a \cdot k$ respectively. An optimal encoding scheme (see Appendix A) would take $\log_2 \sum_{i=0}^{a} \binom{2^k}{i}$ bits per bucket.

[6] There may be fewer on this level in the end of the construction.

This formula for total size, however, is not of much practical value, because buckets should always be aligned and padded to cache line, since we want to characterize cache misses with bucket accesses. A formula that takes this into account is the following (tot_size_a for "total size – aligned"):

$$\begin{aligned}\mathrm{E}\left[\text{tot\_size}_a\right] &= \mathrm{E}\left[\#\text{buckets}\right] \cdot W \cdot \frac{1}{\lfloor\frac{W}{\text{sig\_size}(a,k)+a\cdot r}\rfloor}\\ &= \frac{n\cdot W}{(1-p)b}\cdot\frac{1}{\lfloor\frac{W}{\text{sig\_size}(a,k)+a\cdot r}\rfloor} + \mathcal{O}\left(1\right)\text{ bit}\end{aligned}$$

where W is the cache line size in bits. This formula holds because $\lfloor\frac{W}{\text{sig\_size}(a,k)+a\cdot r}\rfloor$ is the maximum number of buckets that fit into a cache line. That many buckets can therefore be stored together and aligned to cache line boundary, taking W bits in total. For the common special case that only one bucket can fit into a cache line, the formula reduces to

$$\mathrm{E}\left[\text{tot\_size}_1\right] = \mathrm{E}\left[\#\text{buckets}\right]\cdot W = \frac{n\cdot W}{(1-p)b} + \mathcal{O}\left(1\right)\text{ bit}$$

Again, those formulae hold for unbounded CEFS. The corresponding bounded versions with parameter t (t non-PHF levels) are then:

$$\begin{aligned}\mathrm{E}\left[\text{tot\_size}_b\right] &= \mathrm{E}\left[\#\text{buckets}_b\right]\cdot(\text{sig\_size}(a,k)+a\cdot r) + \text{phffs\_size}(np^t,r)\\ &= \frac{n(1-p^t)}{(1-p)b}\cdot(\text{sig\_size}(a,k)+a\cdot r) + \text{phffs\_size}(np^t,r) + \mathcal{O}\left(1\right)\text{ bit}\end{aligned}$$

$$\begin{aligned}\mathrm{E}\left[\text{tot\_size}_{ab}\right] &= \mathrm{E}\left[\#\text{buckets}_b\right]\cdot W\cdot\frac{1}{\lfloor\frac{W}{\text{sig\_size}(a,k)+a\cdot r}\rfloor} + \text{phffs\_size}(np^t,r)\\ &= \frac{n(1-p^t)\cdot W}{(1-p)b}\cdot\frac{1}{\lfloor\frac{W}{\text{sig\_size}(a,k)+a\cdot r}\rfloor} + \text{phffs\_size}(np^t,r) + \mathcal{O}\left(1\right)\text{ bit}\end{aligned}$$

$$\begin{aligned}\mathrm{E}\left[\text{tot\_size}_{1b}\right] &= \mathrm{E}\left[\#\text{buckets}_b\right]\cdot W + \text{phffs\_size}(np^t,r)\\ &= \frac{n(1-p^t)\cdot W}{(1-p)b} + \text{phffs\_size}(np^t,r) + \mathcal{O}\left(1\right)\text{ bit}\end{aligned}$$

The function $\text{phffs\_size}(n,r)$ gives the size of a PHF-based function store for n elements, each having a value of r bits. For a typical implementation of the BPZ Algorithm, $\text{phffs\_size}(n,r) = (2+r)1.23n$.

Finally, Figure 4 shows all Pareto-optimal parameter settings with respect to the expected number of cache misses per query and the space overhead for some typical settings of (t,c,r,W). As shown in the graphic, many parameter settings can offer excellent performance while taking only a small space overhead.

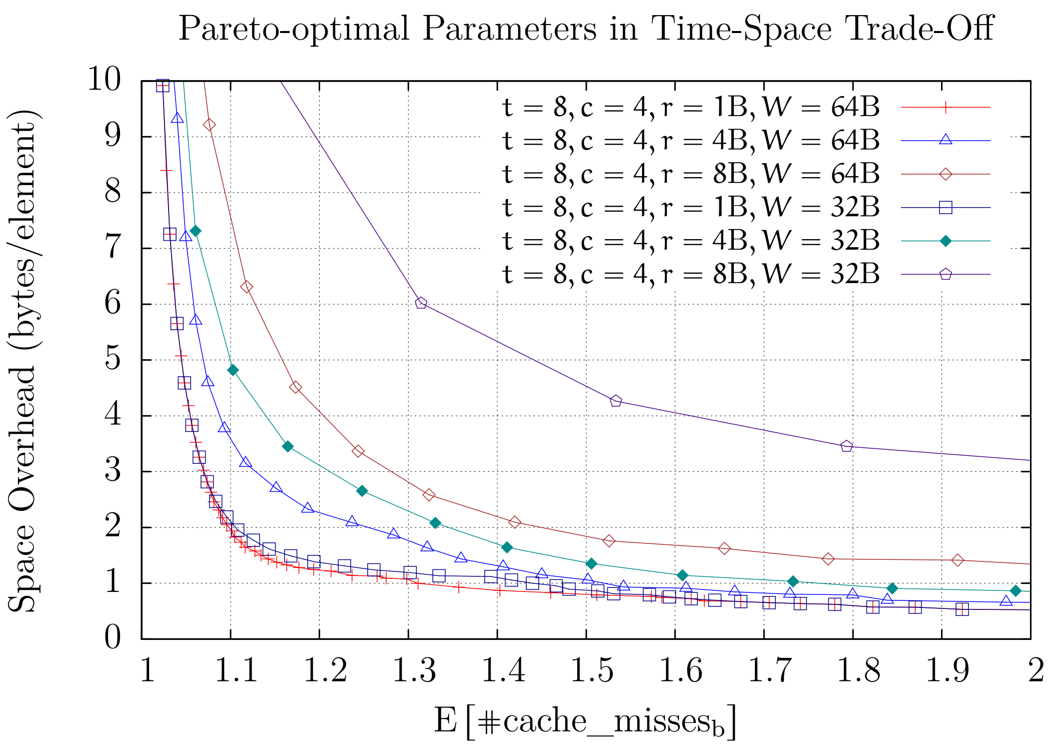

Figure 4: The Pareto-optimal parameter settings with respect to the expected number of cache misses per query and the space overhead. Note the values themselves do not count as overhead. t is the maximum number of non-PHF levels, the last PHF-level answers queries within c cache misses. W is the cache line size. As shown in the graphic, many parameter settings can offer excellent performance while taking only a small space overhead.

5 Experimental Results

From the theoretic point of view, our design goal has been reached: we have a new implementation of function stores that offers excellent balance in the time-space trade-off. However, the theory has yet to be confirmed by experiments. In this section, we present experimental results that demonstrate the time-space balance achieved by CEFS.

Experiments are conducted on an HP Z600 workstation with the following specifications:

OS: Ubuntu 12.04.3 LTS 64 bit.
CPU: 2x Intel® Xeon® Processor X5550 @ 2.66 GHz. Each socket has 4 cores and 8 threads with Hyper-Threading.
Cache Line Size: 64 bytes.
Cache Alignment: 64-byte boundary.
Cache Size: L2 = 256 KB/core; L3 = 8 MB/socket.
RAM: 24 GB.

All source codes are in C++ and are compiled using GNU C++ 4.8.1 with following relevant flags and options:

```
-D_GLIBCXX_PARALLEL -O3 -m64 -msse4.2 -fopenmp -std=c++11 -march=native
```

The sorting in the building procedure has been parallelized with `-D_GLIBCXX_PARALLEL`, a compiler flag offered by GCC that enables parallelism for many algorithms in the STL [22]. Note the parallel sorting is comparison-based, thus is no linear-time algorithm. It has been chosen because of the simplicity to integrate into the test and it beats the (sequential) Counting Sort by far. Other easily parallelizable parts are parallelized with OpenMP.

The number of cache misses is measured with Intel® Performance Counter Monitor (Intel PCM). Cache misses on every core are summed up. This does not impose much inaccuracy since the system is not performing other intensive tasks at the time of the benchmarking. On the contrary, this offers the ability to measure cache misses for multithreaded programs.

5.1 Benchmark Configuration

The CEFS is benchmarked against other competitive implementations of function stores. These include the following:

BPZ: an implementation of the BPZ Algorithm by the author.
CHM-2: an implementation of the CHM Algorithm by the author.
CHM-3: an implementation of the CHM Algorithm by the author.
CMPH-BPZ: an implementation of the BPZ Algorithm from the C Minimal Perfect Hashing Library (CMPH) [7], which is maintained by some of the inventors of these perfect hash functions.
CMPH-CHD-α: an implementation of the CHD Algorithm from the CMPH. α is the load factor of the resulting PHF, affecting only the build time and space overhead.

Algorithm	E [#cache_misses]	Space Overhead (byte)
CEFS-$(13, 8, 32)$	1.053	4.182 $(r = 8\ \text{bit})$
CEFS-$(31, 8, 32)$	1.152	1.378 $(r = 8\ \text{bit})$
CEFS-$(58, 7, 48)$	1.585	0.748 $(r = 8\ \text{bit})$

CEFS-$(7, 7, 12)$	1.061	5.699 $(r = 32$ bit$)$
CEFS-$(13, 7, 12)$	1.237	2.088 $(r = 32$ bit$)$
CEFS-$(19, 6, 14)$	1.502	1.058 $(r = 32$ bit$)$
CEFS-$(4, 7, 6)$	1.076	9.219 $(r = 64$ bit$)$
CEFS-$(7, 6, 7)$	1.244	3.369 $(r = 64$ bit$)$
CEFS-$(10, 6, 7)$	1.526	1.761 $(r = 64$ bit$)$
BPZ	4	$(0.23rn + 2.46n)/8$
CHM-2	2	$(1.09rn)/8$
CHM-3	3	$(0.23rn)/8$
CMPH-BPZ	4	$(0.23rn + 2.46n)8$
CMPH-CHD-0.5	2	$(rn + 0.67n)/8$
CMPH-CHD-0.99	2	$(0.01rn + 2.08n)/8$

Table 1: Algorithms used in the comparison and their properties.

Since in this thesis we are more interested in fast queries in the present thesis, the PHF-based function stores are not implemented with *minimal* perfect hash functions, otherwise more cache misses will be introduced due to accesses in additional rank tables.

Three different Pareto-optimal parameter settings are chosen for CEFS for every different value size (1, 4 or 8 bytes). For every fixed r, the parameters are chosen in such a way that the expected numbers of cache misses per query lie near 1.05, 1.20 and 1.50. We therefore have $3 \times 3 = 9$ CEFS instances. All instances have $t = 8$ non-PHF levels at maximum and a last PHF level implemented with the BPZ Algorithm, which can answer queries within $c = 4$ cache misses. The properties of the used algorithms are summarized in Table 1.

The following two datasets are used in the comparison:

INT. $n = 100\,000\,000$ unique 32-bit integers are generated and used as the keys. Values are random integers of size 1, 4 and 8 bytes.

NGRAMS. These are occurrence statistics on the *n-grams* (i.e. n consecutive words) from Google Books[25], offered by Google. Specifically, 3-grams that start with "na" is used. All entries for the same 3-gram (by years) are combined into a single entry, the count itself serves as the corresponding value, taking the lower bits for smaller values. The combined dataset is randomly permuted. It contains $n = 21\,381\,162$ items. The minimum, maximum and average lengths of the keys are 6, 145 and 23.814, respectively.

The benchmarking procedure is simple: we generate or read the test dataset, construct the various function stores, and query each key once sequentially[7].

All measurements are calculated by taking the average over ten runs, excluding the minimum and the maximum.

5.2 Benchmark Results

The results of the benchmark are presented in this sub-section. They are naturally divided into three parts: the space overhead, the construction algorithm, and the evaluation procedure.

[7]Note that the keys are either random or are permuted randomly in advance.

5.2.1 Space Overhead

Figure 5 shows the space overhead of the data structures. The memory taken by the values themselves is excluded.

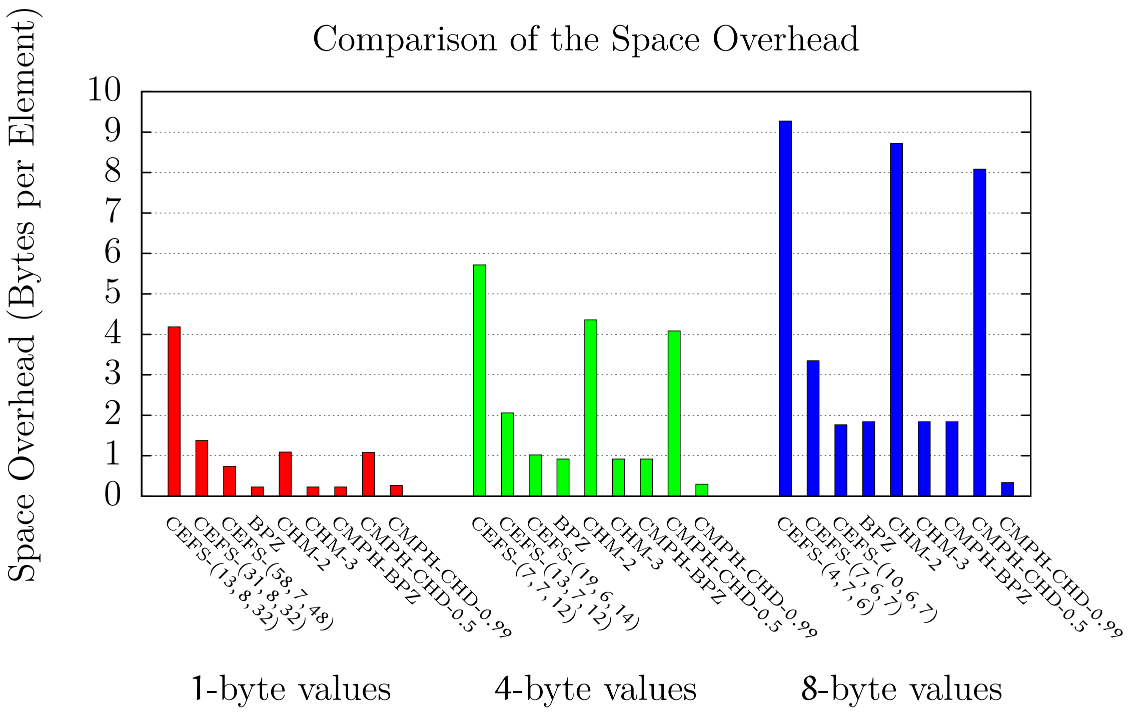

Figure 5: Comparison of the space overhead per element. Values themselves don't count as overhead.

We first note that the space usage of PHF-based function (function stores based on the BPZ and CHD Algorithm) stores are linearly increasing as the number of value bits r increases. For example, the CMPH-CHD-0.5 algorithm has 1 byte overhead for 1-byte values, and 4 bytes for 4-byte values. Generally, that is because PHF-based function stores have an extra value array with a portion of wasted slots. The space usage of those PHFs themselves is independent of r, and the number of wasted slots depends linearly on r. This observation is valid for all those PHF-based function stores, including the apparent exception of CMPH-CHD-0.99: it wastes only 1% of the slots. Therefore the slope of the linear dependence is low, but still linear.

Such a linear dependence is not apparent for CEFS because different parameters are chosen for different r. But since the parameters are chosen in such a way that the expected number of cache misses are at about the same level for different r, a similar tendency of space overhead as r increases can still be observed.

As we can see, CEFS is not always as memory-efficient as other approaches, but there are parameters with which the CEFS can compete with them in space usage while retaining fast evaluations. This is also true for larger values.

5.2.2 Construction

The building times of various data structures are presented in Figure 6. The two bars for each algorithm represent the results for the INT and NGRAMS data sets, respectively. The times are divided by n in order to make the test results also for different datasets comparable.

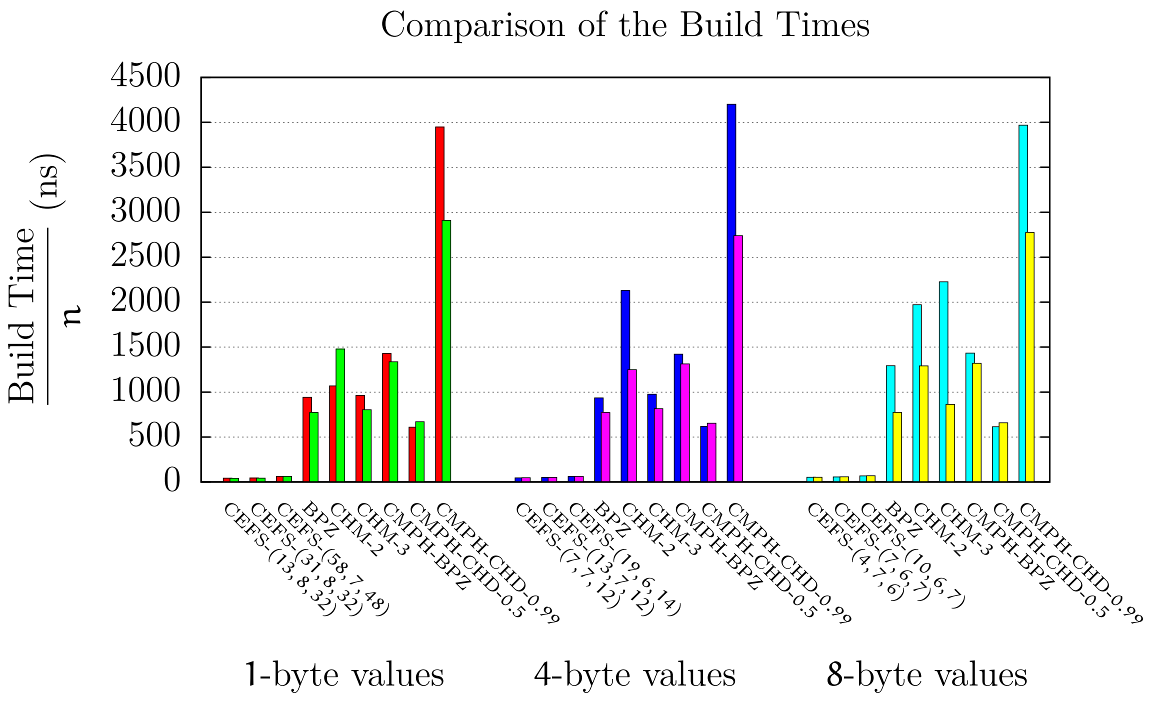

Figure 6: Comparison of building times. The left bar and right bar are running times for INT and NGRAMS, respectively.

One of the attractive features of the CEFS is that the building procedure can be massively parallelized, thus it can be built very efficiently. This feature is demonstrated in the figure. Note how the efficiency of the building procedure of CEFS surpasses all other competitors by at least a magnitude. The other algorithms don't seem easily parallelizable. Even the sequential implementation has been observed to run three times faster than the fastest (CMPH-CHD-0.5) of the other algorithms.

The building times for INT are slightly higher than for NGRAMS for the most algorithms, but the differences between two datasets are not large for most of the times, with the notable exception of CMPH-CHD-0.99 for unknown reasons. The differences for BPZ, CHM-2 and CHM-3 are expected, because a super-linear-time but parallelizable sorting algorithm is used instead of a cache-inefficient linear-time algorithm.

In this perspective, the CEFS shows definite advantage over all other algorithms.

5.2.3 Evaluation

The efficiency of CEFS's evaluation algorithm is also confirmed in the experiments.

We first take a look at the measured cache behaviors of the algorithms for INT and NGRAMS in Figure 7 and Figure 8, respectively. Every algorithm in those graphs has at least two bars in the histogram, the counts of L2 and L3 cache misses respectively. CEFS instances have two more bars which represent the bucket access count and the expected number of cache misses (both per query).

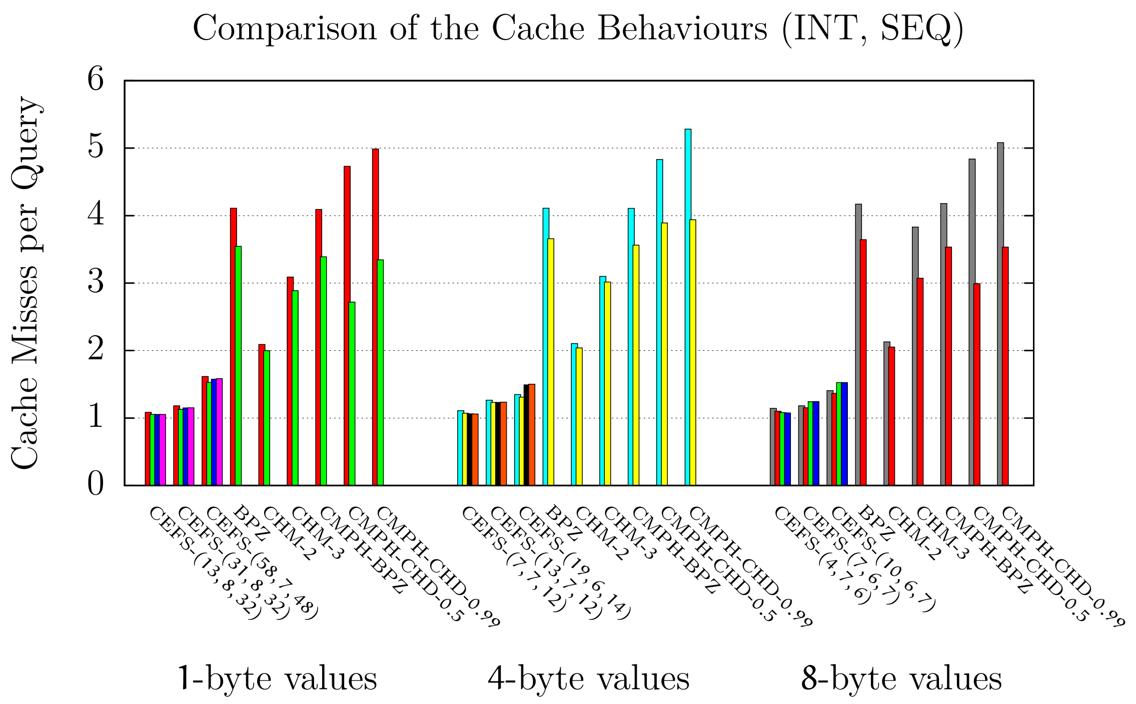

Figure 7: Comparison of the cache behaviors using the INT dataset. The first two bars for each algorithm represent the counts of L2 and L3 cache misses, respectively. CEFS instances have two more bars, the bucket access count and the expected number of cache misses.

For non-PHF-based algorithms, i.e. CEFS and the CHM Algorithm, the difference between the counts of L2 and L3 cache misses is negligible for both datasets. That is because the memory accesses in those algorithms are always carried out in huge arrays, therefore each memory access introduces a cache miss, and each L2 cache miss is also an L3 cache miss, since the L3 cache is still too small to hold any non-negligible portion of the huge arrays.

It is worth noting how the measured cache misses, the measured bucket access count and the calculated expected number of cache misses for CEFS instances coincide almost perfectly[8] at 1.05, 1.20 and 1.5 as expected. This justifies our design rationale: each bucket access is indeed one cache miss. It further confirms the theoretical analysis in Section 4 and indicates that we can use the developed formulae to precisely predict the performance of the algorithm for different parameters, which makes automated parameter selection possible.

The situation is different for the PHF-based function stores. The perfect hash functions themselves are usually very compact, therefore many memory accesses in the internal array of the PHF can be cache *hits*. Only the accesses into the value array are unavoidably cache misses (this is also why the cache misses of CHM-2 and CHM-3 Algorithms lie exactly on the line 2 and 3). This phenomenon is more apparent in the diagram for NGRAMS, which has about 20 million keys. This implies that most PHFs discussed here use only no more than 4 MB, and thus fit theoretically into the L3 cache. The INT dataset has 100 million entries, therefore the PHFs do not fit completely in the cache, but still a portion thereof is in the L3 cache, explaining the difference in the numbers of L2 and L3 cache misses. But even in this case, the CEFS is still in

[8]There are slightly more L2 cache misses than other quantities for the NGRAMS dataset, because the keys themselves are longer and also cause cache misses.

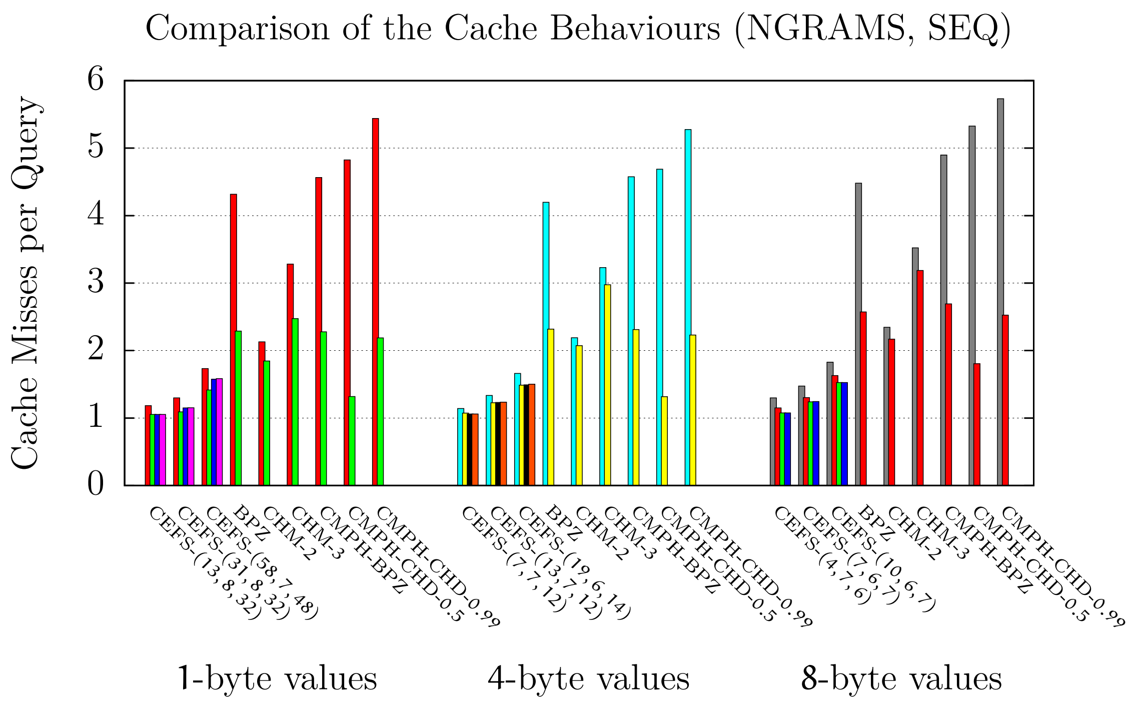

Figure 8: Comparison of the cache behaviors using the NGRAMS dataset. The first two bars for each algorithm represent the counts of L2 and L3 cache misses, respectively. CEFS instances have two more bars, the bucket access count and the expected number of cache misses.

advantage, since a trivial lower bound on the number of cache misses is 1 and CEFS approaches this lower bound. This advantage becomes larger and larger as the number of elements increases and when the data structure is cold.

We now turn to the actually measured time for evaluations in Figure 9. The two bars for each algorithm represent times for INT and NGRAMS, respectively. The tendency shown in the three sub-graphics are almost the same as in Figure 7 and Figure 8. This corresponds to our expectation that the running time for the evaluation is dominated by the cache misses. Note that although the numbers of L3 cache misses of PHF-based function stores in Figure 8 are small (the CMPH-CHD-0.5 Algorithm even has almost as few L3 cache misses as CEFS), the differences in the number of L2 cache misses also make a huge difference in the running time. Therefore we should concentrate more on L2 instead of L3 cache misses if they differ when making choices. For CEFS, however, the L2 and L3 cache misses are identical. The advantage of CEFS is better seen for larger values. The closest competitor is the CHM-2, which guarantees two cache misses in the worst case at the cost of high space overhead and long construction time.

The running times for NGRAMS are slightly longer in Figure 9 for most algorithms because the keys are longer. The reason for the abnormality of CHM-3 Algorithm for the INT dataset and 8-byte values remains unclear.

The figures on the query times and cache behaviors are important because they demonstrate how the theory accords with the experiments. The coincidence of number of cache misses, bucket accesses and the actual running time indicates that the theoretical analysis models the behavior correctly and the calculation is quite accurate. They have thereby shown that CEFS is very

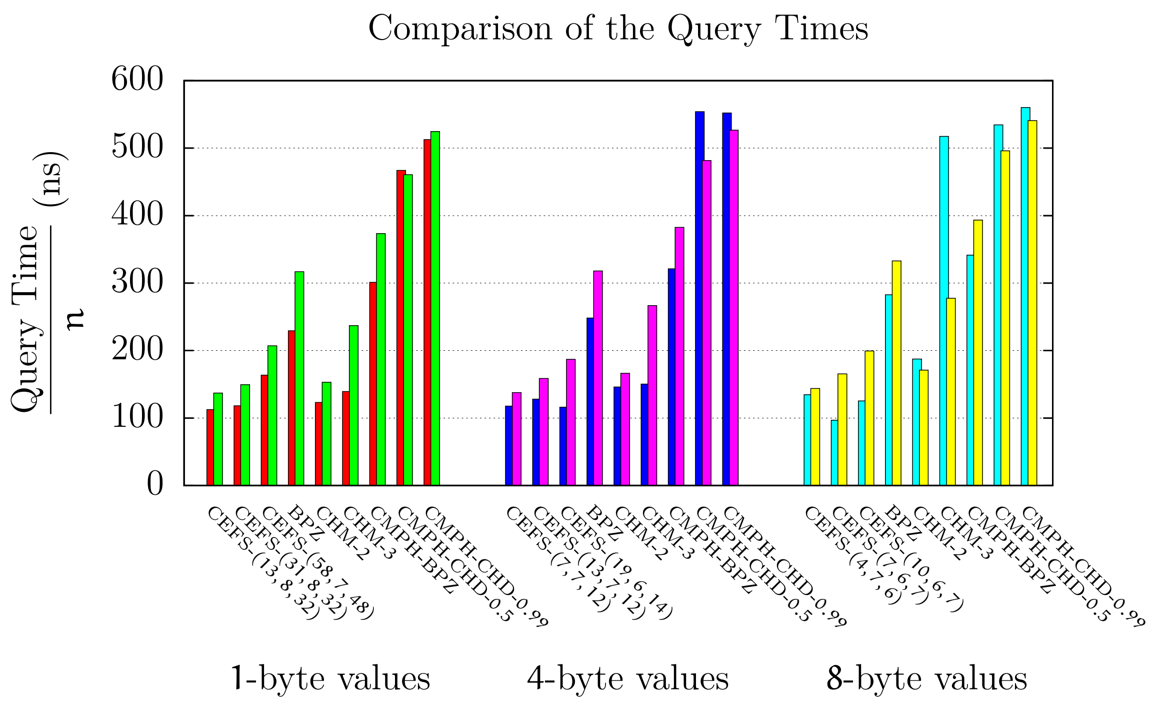

Figure 9: Comparison of evaluation times. The left bar and right bar are running times for INT and NGRAMS, respectively.

efficient in practice as well as in theory.

6 Conclusion

6.1 Summary

In the present thesis a conceptually very simple yet very efficient implementation of function stores has been proposed, i.e. the Cache-Efficient Function Stores (CEFS).

The data structure can be built in $\mathcal{O}(n)$ expected time for n elements. The building procedure runs faster than any known methods and improves upon known methods by at least a magnitude if the building algorithm is parallelized.

The evaluation takes worst-case constant time. If parameters are properly chosen and a value itself is not too large, the expected number of cache misses per query can be made to just slightly above one with small space overhead. The space requirement apart from the memory required for the values themselves is small. The time and space properties for different parameters can be precisely predicted with the developed formulae in Section 4 beforehand.

Experimental results have also demonstrated the efficiency of the new data structure with significant improvement with respect to construction time and evaluation time over other known methods.

6.2 Open Problems for Future Research

The CEFS, however, also has drawbacks and unsolved problems associated with it. This last section names some of them. The list could serve as a brief list of possible directions for future research.

Problem 1. Drawbacks caused by not storing keys.

The keys are not stored along with values, which saves much space and many comparisons. However, there are things that can (possibly) only be solved with keys. For example listing all (key, value) pairs is not possible in a CEFS. Another problem is that we cannot always identify keys that were not present at creation of the CEFS. These problems do not seem resolvable without explicitly storing the keys.

Problem 2. Is it possible to make CEFS dynamic?

The data structure is by definition static. No insertion or deletion is allowed. This is no big problem for applications where such updates are rare, but inapplicable if they are not. Note, however, that changing the value of a present key is easy in CEFS as long as the last PHF-level supports that.

Problem 3. Can the falling proportion $p(n, m, a, d)$ be calculated or approximated with simpler expressions?

The formulae in Section 4 are exact. This is important in theory because we can precisely predict the performance of the data structure without having to actually build it. However, it is not that useful in practice due to its tediousness and high computational complexity.

Problem 4. Can CEFS be sped up for a specific distribution of queries given beforehand?

In the analysis of CEFS (Section 4), the running time of queries are made independent of the keys themselves due to the random selection of hash functions. The CEFS would offer even better performance if it can smartly distribute all keys into levels according to their access probabilities, i.e. frequently used keys are placed more shallowly.

In fact, this problem can already be partially solved. At the capacity compliance stage in Section 2.3, we actually have certain degree of freedom at choosing elements that shall fall into next levels. Choosing less frequently accessed ones would result in better performance. The problem here is to precisely model the benefit of this strategy.

Problem 5. Does it make sense to have different parameters for different levels?

In the CEFS, all levels share the same set of parameters. This has the advantage that parameters do not have to be stored in every level and that the calculation of overall performance involves (almost) only the same falling proportion. But having different parameters according to the number of elements in each level *might* offer better performance.

Problem 6. Can the same idea be applied to other data structures?

An essential idea in the CEFS is to use short signatures to deterministically and correctly identify keys. Such an idea might also be applicable to other data structures like Cuckoo Hashing [12, 14, 13] so that no keys have to be stored along with the values.

References

[1] L. Comtet, *Advanced Combinatorics: The Art of Finite and Infinite Expansions.* D. Reidel Publishing Company, 1974. (Pages iii, 14).

[2] F. T. Howard, "Associated Stirling numbers," *Fibonacci Quart.*, 1980. (Pages iii, 13, 14).

[3] OEIS Foundation Inc., "The On-Line Encyclopedia of Integer Sequences," A008299, 2011, triangle of associated Stirling numbers of second kind. (Pages iii, 13, 14).

[4] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," in *Algorithms and Data Structures.* Springer, 2007, pp. 139–150. (Pages iv, 2).

[5] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, "Hash, displace, and compress," in *Algorithms-ESA 2009.* Springer, 2009, pp. 682–693. (Pages iv, 2, 8).

[6] Z. J. Czech, G. Havas, and B. S. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, vol. 43, no. 5, pp. 257–264, 1992. (Pages iv, 3).

[7] D. de Castro Reis, D. Belazzougui, F. C. Botelho, and N. Ziviani, "CMPH – C Minimal Perfect Hashing Library." [Online]. Available: http://cmph.sourceforge.net (Pages iv, 21).

[8] Intel Corporation, "Intel® Performance Counter Monitor – a better way to measure cpu utilization." [Online]. Available: http://www.intel.com/software/pcm (Page iv).

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. (Pages 1, 2).

[10] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2004, pp. 30–39. (Pages 1, 3).

[11] M. Dietzfelbinger and R. Pagh, "Succinct data structures for retrieval and approximate membership," in *Automata, Languages and Programming.* Springer, 2008, pp. 385–396. (Pages 1, 3).

[12] R. Pagh and F. Rodler, "Cuckoo hashing," in *Algorithms – ESA 2001.* Springer Berlin Heidelberg, 2001, vol. 2161, pp. 121–133. (Pages 2, 29).

[13] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," in *In STACS*, 2003, pp. 271–282. (Pages 2, 29).

[14] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," in *Automata, Languages and Programming.* Springer Berlin Heidelberg, 2005, vol. 3580, pp. 166–178. (Pages 2, 29).

[15] B. Jenkins, "Algorithm alley: Hash functions," *Dr. Dobb's Journal*, 1997. (Page 8).

[16] F. C. Botelho and N. Ziviani, "External perfect hashing for very large key sets," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management.* ACM, 2007, pp. 653–662. (Page 8).

[17] M. Dietzfelbinger and M. Rink, "Applications of a splitting trick," in *Automata, Languages and Programming.* Springer, 2009, pp. 354–365. (Page 8).

[18] Intel Corporation, *Intel®64 and IA-32 Architectures Software Developer's Manual*, 2013. (Pages 9, 10).

[19] Advanced Micro Devices Inc., *AMD64 Architecture Programmer's Manual*, 2013, vol. 3: General-Purpose and System Instructions. (Page 9).

[20] Advanced Micro Devices Inc., *AMD64 Architecture Programmer's Manual*, 2013, vol. 2: System Programming. (Page 10).

[21] B. Stroustrup, *The C++ Programming Language : [C++11]*, 4th ed. Addison-Wesley, 2013. (Page 10).

[22] Free Software Foundation, *The GNU C++ Library Manual*, 2013. (Pages 11, 21).

[23] M. H. DeGroot and M. J. Schervish, *Probability and statistics*, 3rd ed., ser. Pearson education. Addison-Wesley, 2002. (Page 15).

[24] E. Zakon, B. Lucier, and T. Zakon, *Mathematical Analysis*, ser. The Zakon Series on Mathematical Analysis. Trillia Group, 2004. (Pages 15, 17).

[25] Google, "Google books Ngram Viewer." [Online]. Available: http://storage.googleapis.com/books/ngrams/books/datasetsv2.html (Page 22).

[26] Creative Commons, "Attribution 3.0 Unported License." [Online]. Available: http://creativecommons.org/licenses/by/3.0

Appendix A An Optimal Signature Encoding Scheme

A simple but relatively efficient method to encode and decode the signatures in a bucket is presented. The encoding scheme is optimal in the sense that every possible signature sequence has a unique encoding, and every possible code represents a signature sequence. That is to say, there exists a bijection between possible codes and sequences of signatures. Only a small table which fits in L1 cache is needed.

The following notations are needed in this section:

a:	bucket capacity.
k:	number of signature bits.
d:	the upper bound of the signatures, defined as 2^k. All signatures fall into $[0..d-1]$.
p:	the signature sequence, sorted in ascending order. $p[i]$ represents the i-th element. Indices begin with 0.
$p_{\mathtt{bin}}$:	The corresponding (length d) binary number to p, produced by treating elements in p as the positions of ones. 0 in the sequence represents the most significant bit. $p_{\mathtt{bin}}[i]$ is the i-th bit of the number, counting from the most significant bit.
m:	the length of the signature sequence. $m \leqslant a$ holds.
pop(x):	the number of ones in binary number x.

A.1 The Scheme

All possible signature sequences with up to a elements are sorted according to the following comparator (most significant first):

(i) The sequence with less elements is listed first.
(ii) Write the sequences in binary representations as described above. The sequence producing the smaller number is listed first.

The code for a signature sequence is just its index in the sorted list of all possible sequences. A sample of the encoding for $a = k = 4$ is presented in Table A.1.

Code	Sequence	Binary Number
0	[]	0000000000000000
1	[15]	000000000000000**1**
2	[14]	00000000000000**1**0
⋮	⋮	⋮
15	[1]	0**1**00000000000000
16	[0]	**1**000000000000000
17	[14, 15]	00000000000000**11**
18	[13, 15]	0000000000000**1**0**1**
19	[13, 14]	0000000000000**11**0
20	[12, 15]	000000000000**1**00**1**
⋮	⋮	⋮
135	[0, 2]	**1**0**1**0000000000000
136	[0, 1]	**11**00000000000000
137	[13, 14, 15]	0000000000000**111**
138	[12, 14, 15]	000000000000**1**0**11**

⋮	⋮	⋮
695	[0, 1, 3]	**1101**000000000000
696	[0, 1, 2]	**111**0000000000000
697	[12, 13, 14, 15]	000000000000**1111**
698	[11, 13, 14, 15]	00000000000**10111**
⋮	⋮	⋮
2515	[0, 1, 2, 4]	**11101**00000000000
2516	[0, 1, 2, 3]	**1111**000000000000

Table A.1: Sample encoding

A.2 The Algorithm

A.2.1 Encoding

The number of sequences with less elements than p is simply $\sum_{i=0}^{m-1}\binom{d}{i}$, so we focus on getting the index of sequence p among all sequences with exactly m elements. That is to count the number of binary numbers with m ones that are smaller than p_{bin}.

Consider a number $q_{bin} < p_{bin}$. There must be a position $0 \leqslant pos < d$, so that p_{bin} and q_{bin} are identical for the bits before pos and are different at position pos, where $p_{bin}[pos] = 1$, $q_{bin}[pos] = 0$. Therefore every such q_{bin} can be generated by the following method: set $p_{bin}[pos]$ to 0, and then distribute $(m - \text{pop}(p_{bin}[0..pos-1]))$ ones[9] freely in positions $pos+1$ to $d-1$. It is also obvious that every generated number is smaller than p_{bin}.

For every possible pos (with $p_{bin}[pos] = 1$), which is by definition an element in p (we call it $p[i]$), we count the number of binary numbers having m ones that are the same with p_{bin} until position pos (counting from most significant bit). According to the above method, there are exactly $\binom{d-1-(pos+1)+1}{m-\text{pop}(p_{bin}[0..pos-1])} = \binom{d-1-p[i]}{m-i}$ of such numbers. Summing up those counts gives the number of sequences with m elements before p in the list.

The final code of p is then given by

$$\text{code}(p) = \sum_{i=0}^{m-1}\binom{d}{i} + \sum_{i=0}^{m-1}\binom{d-1-p[i]}{m-i}$$

A direct implementation could use $2m$ look-ups into a d-by-a table storing the binomial coefficients. This can be reduced to m look-ups by exploiting the fact $\binom{d}{i} = \binom{d}{i-1}\cdot\frac{d-i+1}{i}$ for $i > 0$. So those m look-ups on the left side can be eliminated. This result in an implementation with $\mathcal{O}(m)$ integer arithmetics and m table look-ups. Whether this is an improvement remains unclear though, because the table is so small that it fits well in L1 cache (a multiplication and a division vs. a table look-up). The look-ups on the right side seem hard to eliminate without taking $\mathcal{O}(d)$ time.

A.2.2 Decoding

Given a code c, we first determine the length ($=: m$) of the encoded signature sequence ($=: p$). This is done by simply finding the first m such that $\sum_{i=0}^{m}\binom{d}{i} > c$. We then determine the bits

[9]Because q_{bin} has m ones in total, and it has already used pop($p_{bin}[0..pos-1]$).

in p_{bin} one by one, from the most significant bit.

To determine bit i, we first make a *hypothesis* that this bit was 0. We know from above that there are $\binom{d-1-i}{m-\mathtt{settled}}$ satisfying given conditions, where $\mathtt{settled}$ is the number of ones we have used in positions $0..i-1$. If this count is larger than c, we know the hypothesis was correct, this bit was indeed 0. Otherwise, that bit was 1. We then note this down, subtract this count from c, and go to the next bit.

This decoding algorithm takes $\mathcal{O}(d)$ time in total and $a + d$ look-ups for the both stages (determining m and then determining every bit). Unlike the encoding algorithm, the look-ups here can be completely eliminated using similar facts as above, because the transition from one binomial coefficient is always to an *adjacent* binomial coefficient in the sense that either the top or the bottom number is incremented or decremented by 1. Again, it is unclear whether this is an improvement.