SicHash – Small Irregular Cuckoo Tables for Perfect Hashing

Hans-Peter Lehmann* Peter Sanders† Stefan Walzer‡

Abstract

A Perfect Hash Function (PHF) is a hash function that has no collisions on a given input set. PHFs can be used for space efficient storage of data in an array, or for determining a compact representative of each object in the set. In this paper, we present the PHF construction algorithm SicHash – Small Irregular Cuckoo Tables for Perfect Hashing. At its core, SicHash uses a known technique: it places objects in a cuckoo hash table and then stores the final hash function choice of each object in a retrieval data structure. We combine the idea with *irregular* cuckoo hashing, where different objects can have a different number of hash functions. Additionally, we use many small tables that we *overload* beyond their asymptotic maximum load factor. The most space efficient competitors often use brute force methods to determine the PHFs. SicHash provides a more direct construction algorithm that only rarely needs to re-compute parts. Our implementation improves the state of the art in terms of space usage versus construction time for a wide range of configurations. For some configurations, SicHash is up to 4.3 times faster than the next best competitor. At the same time, it provides very fast queries.

1 Introduction

A Perfect Hash Function (PHF) is a hash function that does not have collisions on a given set S of objects, i.e., it is injective. In this paper, we call the number of objects N and the size of the output range M. A PHF with $M = N$ is called *Minimal* Perfect Hash Function (MPHF). For M close to N, it is likely that an ordinary hash function has collisions. It is therefore necessary to store additional information, specific to the set S, that help to avoid these collisions. The lower space bound for an MPHF is 1.44 bits/object [2]. PHFs can be represented with less space, depending on the load factor N/M. In the literature, load factors between 0.8 and 1.0 are common [2, 38].

PHFs can be used to implement a hash table where each query only needs to access one single cell. When we know that the hash table is never queried for objects $\notin S$, table cells can store the plain payload data without keys. This results in a retrieval data structure that can be updated efficiently. Refer to Section 2 for more details about retrieval data structures and an introduction to cuckoo hashing, which is a main building block of SicHash. There is a wide range of PHF construction algorithms, which we review extensively in Section 3.

The basic idea of SicHash is to distribute the input objects to a number of small buckets and build a cuckoo hash table in each. In cuckoo hashing, each object has a number of different positions it can be stored in, determined by different hash functions. We then use a retrieval data structure to store which of its candidate positions was finally used for each object. To achieve better space efficiency, we use *irregular* cuckoo hashing [10], where an additional hash function determines how many candidate positions each object has. The reason for constructing *small* tables is the use of overloading, which we describe in Section 4. In Section 5, we explain SicHash in detail, giving construction and query algorithms. Enhancements of the basic scheme, including one that produces MPHFs, are given in Section 6. We analyze SicHash in Section 7. We then provide an extensive experimental evaluation in Section 8, comparing it with a wide range of competitors. In Section 9, we summarize the results and discuss possible future work.

Comparison with Previous Approaches. The most space-efficient previous algorithms perform brute-force search as a core step to determine a perfect hash function. In particular, both CHD [2] and RecSplit [15] at some point try out random hash functions (on sub-problems) until one happens to be injective. Refer to Section 3 for details. Our construction is more directed than this because it constructs cuckoo hash tables as its base case, which is possible in polynomial time. The directedness is also visible in the experiments, where our method can construct PHFs with the same space requirements up to 4.3 times faster than the competitors.

Our Contribution. We combine and refine several known ideas in a novel way leading to excellent space–construction time trade-offs while using very low query time. We base SicHash on the known idea of PHF generation through cuckoo hashing. We use *irregular*

*Karlsruhe Institute of Technology, Germany.
†Karlsruhe Institute of Technology, Germany.
‡Cologne University, Germany.

Copyright © 2023
Copyright for this paper is retained by the authors

cuckoo hashing [10], which was previously considered for reducing search time. For that application, it was of little help apart from allowing to interpolate between two integer numbers of hash functions. In contrast, for our application to reduce space, it is helpful even when the average number of hash functions already is an integer. Space is further reduced using the novel idea to *overload* the cuckoo hash tables, i.e., to load them with more objects than would be possible in an asymptotic sense, exploiting that the tables are small. All this keeps the queries extremely simple – basically the cost for a single access to a retrieval data structure. This further profits from recent advances on fast static retrieval data structures with virtually no space overhead [13]. In turn, our PHFs can be used to obtain improved *updateable* retrieval data structures as discussed before.

2 Preliminaries

Cuckoo Hashing. Cuckoo hashing [37] is a well known approach to hash tables with open addressing. In a basic cuckoo hash table, each object can be placed in one of two cells, determined by two hash functions. Queries load the two candidate cells and compare both objects. Insertion applies one of the hash functions and places the new object in the corresponding cell. If the cell is already occupied, the object previously placed in that cell is pushed out and is recursively inserted using its other hash function.

Instead of locating each object in one of two cells, the idea can be generalized to d cells [17] by using d hash functions. In *irregular* cuckoo hash tables, different objects can have a different number of choices [10]. For example, some percentage of the objects get d_1 choices, some d_2 choices, and some d_3 choices. Averaging over the d_i, the method enables d-ary cuckoo hashing with non-integer d and higher load factors than a simple interpolation between two ordinary cuckoo hash tables [10].

Retrieval Data Structures. A *retrieval data structure* or *static function* on a set S of objects describes a function $f : S \to \{0, 1\}^r$ that returns a specific r-bit value for each object. Applying the function on an object not in S can return an arbitrary value. The lower bound of the space requirement of a retrieval data structure is rN bits. The best retrieval data structures now come very close to the lower bound (around 1% overhead) and are also quite fast [13].

PHF Construction by Cuckoo Hashing. To the best of our knowledge, constructing PHFs through cuckoo hashing was only mentioned very briefly before [13]. In this paragraph, we give a more detailed and intuitive introduction to the idea. A related idea is the construction of PHFs by solving a matching as described by, e.g., Botelho et al. [5] and Navarro [36, Section 4.5.3].

Assume that all input objects are inserted into a d-ary cuckoo hash table. The table then implicitly describes an injective mapping from objects to table cells, because each cell only stores one object. For perfect hashing, we are not interested in storing the objects themselves but only in mapping objects to numbers, e.g., table cell indices. Because each object can only be placed in d cells using d hash functions h_i, we can remember the placement of each object by simply storing which of the hash functions was finally used to place the object. We can do that using only about $\log d$ bits[1] per object by constructing a retrieval data structure. A query for an object x then retrieves the hash function index $i(x)$ and executes $h_{i(x)}(x)$ to obtain a perfect hash function.

Elias-Fano Coding. Elias-Fano Coding [14, 16] is a way to efficiently store a monotonic sequence of N integers. It consists of two data structures, a bit vector H, and an array L. An item at position i is split into two parts. The $\log N$ upper bits u are stored as a 1-bit in $H[i + u]$. The remaining lower bits are directly stored in L. Items can be accessed in constant time by finding the i-th 1-bit in H using a $select_1$ data structure and by looking up the lower bits in L. The space usage of an Elias-Fano coded sequence is $2N + N\lceil \log U/N \rceil$ bits, where U is the maximum value of an item.

Golomb-Rice Coding. Golomb coding [23] with parameter k can be used to store a sequence of integers that have a geometric distribution. The idea is to store each integer x as a quotient $q = \lfloor x/k \rfloor$ in unary coding and a remainder $x - qk$ in truncated binary coding. Rice coding [39] is Golomb coding where k is a power of 2. This makes arithmetics more efficient and simplifies storing the remainder to a normal array with binary coding. Items can be accessed in constant time by looking up the array and reconstructing the quotient using a $select_1$ query.

3 Related Work

In the following, we first describe variants and enhancements of cuckoo hashing from the literature. Afterwards, we introduce existing PHFs, most of which we later include in our experimental evaluation (see Section 8).

3.1 Cuckoo Hashing.

After describing variants of cuckoo hashing, we describe construction algorithms and load thresholds.

Variants. Higher maximum load factors can be achieved by making the cells larger, so that they hold more than one object [12]. When then allowing the cells to overlap [31], even higher load factors are possible [42]. For our application to perfect hashing, we only consider cells of size 1. In external memory, I/Os can be reduced

[1] Throughout this paper, $\log x$ stands for $\log_2 x$.

Copyright © 2023
Copyright for this paper is retained by the authors

by ensuring that most candidate cells are selected on the same memory page [11]. Maintaining two tables [37] of asymmetric size [30] can improve the search time because more objects can be placed using their first hash function. Giving each object d instead of 2 choices [17] increases the maximum load factor. *Irregular* cuckoo hashing [10] uses an additional hash function to determine how many choices each object has. A similar idea can also be found in coding theory, where each message bit is covered by an irregular number of check bits [34]. Specifically, the probability that a message bit is covered by i check bits is proportional to $1/i$. Another related result is the weighted Bloom filter [7], where objects get a different number of hash functions (and therefore false positive probability) based on their query frequency and membership likelihood.

Construction. The enhancement to d-ary cuckoo hashing [17] makes insertions more complex because it is no longer clear which of the alternative cells to displace objects to. Common ways to perform insertion are to find a shortest move sequence by performing *breadth-first-search (BFS)* in a graph defining possible object moves or by performing a *random walk* in that graph. Both approaches need constant expected time when the table is not too highly loaded [17, 44, 21, 19, 26, 27].

In this paper, we are interested in the static case, where all objects to be stored are known from the start. In that case, it is also possible to construct the whole hash table at once instead of using incremental insertions. Let us model the cuckoo hash table as a bipartite graph. The first set of graph vertices is simply the set of input objects and the second set represents the table cells. Edges connect each object to its candidate cells, as determined by the d hash functions. A matching of size N then gives a collision-free assignment from objects to table cells. This can be calculated using, for example, the Hopcroft-Karp-Karzanov algorithm [24] or the LSA algorithm [26, 27].

Load Thresholds and Space Usage. The *load threshold* is the load factor N/M such that the probability of successful construction tends to 1 for smaller load factors and to 0 for larger load factors for $M \to \infty$. Classic cuckoo hashing with $d = 2$ hash functions has an asymptotic load threshold of 0.5. Using $d = 4$ hash functions already increases the load threshold to 0.9768 [18, 43]. In our construction, the load factor of the PHF equals the load factor of the cuckoo hash table, and the storage space is determined by the number of hash functions d. This means that a PHF from binary cuckoo hashing with a load factor of 0.5 can be represented using $\log 2 = 1$ bit per object. A PHF with a load factor of 0.9768 can be implemented using 2 bits per object.

Ref. [10] gives load thresholds for irregular cuckoo hashing, depending on the distribution of how many hash function choices each object has. For any desired average number of hash functions $d' \in \mathbb{R}$, the best load threshold is achieved by assigning each object either $\lfloor d' \rfloor$ or $\lceil d' \rceil$ hash functions [10]. As we will see in Section 4, this is not the case in the context of PHFs because we are looking at the storage space instead of the average value of the hash function indices.

3.2 Perfect Hashing. The perfect hashing problem has already been considered since the 1970s [41, 25, 6], and is still an active area of research. In the following paragraphs, we describe more recent papers.

Order-Preserving. An order-preserving PHF maintains the order that the input objects are given in. CHM [9] and BMZ [3] construct an undirected graph with edges $\{(h_1(x), h_2(x)) | x \in S\}$. They then assign a number to each vertex, such that for each edge, the sum of numbers stored in adjacent vertices gives the desired PHF value. This can be done by assigning 0 to an arbitrary vertex and then performing depth-first-search to assign all neighbors by simple subtraction. CHM and BMZ store $2.09N$ and $1.15N$ integer numbers, respectively, therefore needing $O(N \log(N))$ space. Note that space near $N \log N$ can also be achieved by explicitly storing the rank of the objects in a retrieval data structure.

BDZ. In the BDZ algorithm [5], also called RAM algorithm or BPZ algorithm, each input object is mapped to an edge in a random hypergraph using d independent hash functions h_i. The hypergraph needs to be peelable[2] in order to continue. By peeling the graph, BDZ determines $i(x)$, such that $h_{i(x)}(x)$ is unique for each object x. It then uses a linear equation system to determine a function g, such that $i(x) = \left(\sum_{0 \le i < d} g(h_i(x))\right) \bmod d$.

Even though our presentation using cuckoo hashing sounds different, SicHash is similar to this idea. The BDZ algorithm's task of finding a unique $h_{i(x)}(x)$ for each object can be interpreted as placing the objects in a cuckoo hash table. The function g serves as a retrieval data structure that maps each object to a hash function index $i(x)$. The most important difference is that the BDZ algorithm couples retrieval and object placement by using the same set of hash functions. In particular, g is evaluated for the entire range M, so the space to store g depends on M. SicHash, in contrast, separates the two tasks of object placement and hash function retrieval. This enables using a retrieval data structure of size N instead. Moreover, SicHash uses irregular cuckoo hashing, which cannot be represented efficiently with the integrated retrieval data structure of BDZ. Finally, SicHash does not depend on peelability.

[2]Possibility of obtaining a graph without edges by iteratively taking away edges that contain a vertex with degree 1 [5, 43].

Copyright © 2023
Copyright for this paper is retained by the authors

WBPM. Weaver et al. [45] describe an algorithm for calculating MPHFs that is based on weighted bipartite matchings (WBPM). The left vertex set of the bipartite graph is determined by the $M = N$ input objects and the right set is determined by the N possible hash values. The edges are determined by applying $O(\log(N))$ hash functions to each object, where an edge determined from the i-th hash function has weight i. The weighted matching can be solved with a weight of $1.83N$, giving an assignment from objects to hash values. For storing which hash function to use for each object, WBPM uses a 1-bit retrieval data structure. The keys to the retrieval data structure are tuples of object and hash function index. The stored value is 1 for the hash function to finally be used, and 0 for all smaller indices. The weight of $1.83N$ therefore also equals the space usage of the final data structure, except for overheads like prefix sums due to bucketing.

SicHash uses a similar structure but simplifies each of the ingredients. Instead of a *weighted* bipartite matching, SicHash (implicitly) solves a *non-weighted* bipartite matching by constructing a cuckoo hash table. Instead of querying a 1-bit retrieval data structure multiple times for each hash function evaluation, SicHash only performs a single query to a retrieval data structure. While WBPM constructs a retrieval data structure consisting of $1.83N$ objects, SicHash generates retrieval data structures with a total of N objects, which makes the construction faster. While WBPM's space usage is competitive for MPHFs, constructing a non-minimal PHF is less efficient. With a load factor of 0.85, for example, SicHash achieves a space usage of $1.43N$ bits, while our preliminary experiments show that a matching like above has a weight of $1.54N$. This stems from the fact that SicHash stores the selected hash function index using binary code, while WBPM effectively uses unary code.

CHD [2] / FCH [20] / PTHash [38]. The basic idea of this approach is to hash the input objects to a number of buckets with expected equal size. After sorting the buckets by their size, the methods search for a hash function that stores the entire bucket in the final output domain without causing collisions. CHD [2] tries out hash functions linearly, so that the data to store for each bucket can be compressed efficiently. With a load factor of 81%, CHD can construct a PHF using 1.4 bits per object. With a load factor of 99%, it achieves 1.98 bits per object. In FCH [20], the bucket sizes are asymmetric – using the default parameters, 60% of the objects are mapped to 30% of the buckets. After hashing the objects of a bucket, it searches for a rotation value that (mod M) places all objects into the output range without collisions. FCH produces MPHFs with about 2 bits per object. PTHash [38] combines the two ideas by using asymmetric buckets and trying hash functions linearly to enable a compressible representation. To speed up searching for the hash function in each bucket, it first generates a non-minimal PHF. Instead of using the well known *rank trick* [4] to convert a PHF to an MPHF, PTHash uses a new approach that enables faster queries: We can interpret the values in the output range M that no object is mapped to as *holes*. PTHash now stores an array of size $M - N$ that re-maps each object with a hash value $> N$ to one of the holes. Given that the holes are a monotonic sequence of integers, the array can be compressed with Elias-Fano coding. Because most objects are mapped to values $\leq N$, the expensive lookups into the Elias-Fano sequence are only rarely needed.

MeraculousHash [8] / FiPHa [35] / BBHash [33]. This approach hashes the objects to γN buckets, for $\gamma \in \mathbb{R}, \gamma \geq 1$. If a bucket has exactly one object mapped to it, that mapping is injective. Objects from buckets that hold more than one object are bumped to the next layer of the same data structure. For each layer, a bit vector of size γN indicates which buckets had stored a single object, e.g., where the recursive query can stop. Together with a rank data structure, that bit vector can also be used to make the resulting hash function minimal. This approach allows fast construction and queries but needs space at least e bits per object [35] – more than 4 for really good speed.

RecSplit. RecSplit [15] distributes all objects to buckets of expected equal size. Within each bucket, RecSplit first searches for a hash function with binary output that splits the set of objects in half.[3] This is repeated recursively until the set of objects has a small, configurable size. Usual values for the leaf size are about 8–12 objects. At a leaf, RecSplit then tries out hash functions until it finds one that is a bijection. For each bucket, RecSplit stores the hash function seeds that make up the splitting tree, the seeds for each leaf, and a prefix sum of the bucket sizes. There are configurations that need only 1.56 bits per object.

4 Overloading

In cuckoo hashing, the load threshold of a table is a widely studied subject (see Section 3.1). For example, it is well known that a table with $d = 2$ hash functions has a *load threshold* of 50%. For $M \to \infty$, the probability of successful table construction with load factor $> 50\%$ approaches 0, while it approaches 1 for load factors $< 50\%$. Let us now look at a very small table of size $M = 3$ storing $N = 2$ objects. This table has a load factor of $\approx 66\%$, which is more than the load threshold. Still, the probability of successful construction, i.e., not all four

[3] The fanout is larger for the bottommost recursion layers.

Copyright © 2023
Copyright for this paper is retained by the authors

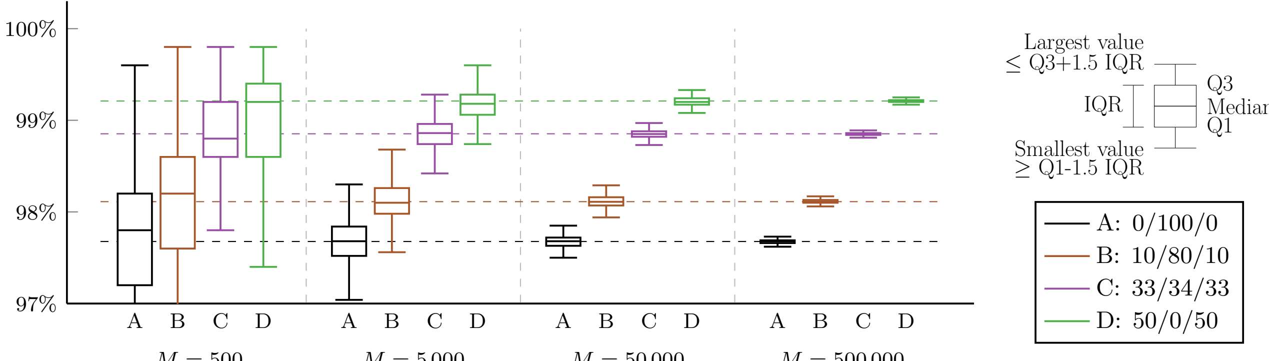

Figure 1: Achieved load factors when running different irregular cuckoo hashing configurations, which all need the same storage space (2 bits). The configurations are described by the percentages of objects with 2/4/8 choices, having a space usage of 1/2/3 bits, respectively. The configuration 0/100/0 refers to ordinary 4-ary cuckoo hashing. Horizontal lines indicate numerically calculated load thresholds (see Section 7).

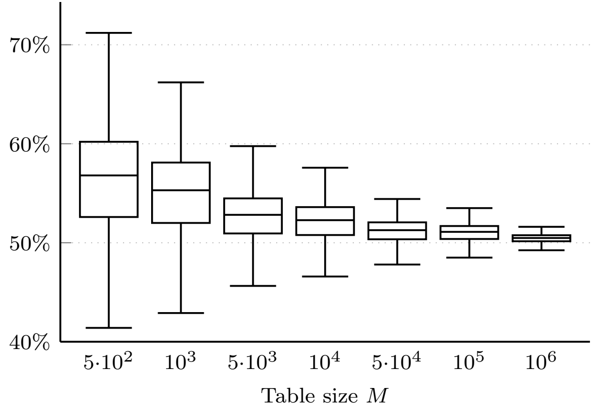

Figure 2: Achieved load factors when constructing ordinary, binary cuckoo hash tables of different sizes. The median of small tables is significantly higher than the asymptotic load threshold (50%).

hash function values being the same, is $1-3\cdot(1/3)^4 \approx 88\%$. This shows that the load factors can be considerably higher than the asymptotic limits when using small tables. We call a table that contains more objects than the asymptotic limit *overloaded*.

To experimentally illustrate the achievable load factors, we incrementally construct cuckoo hash tables and record the load factors at which the insertion finally fails. Figure 1 gives a box plot of the achieved load factors. It shows three fundamental observations that we use in SicHash to increase the load factor while decreasing the amount of memory needed.

(1) Variance. Unsurprisingly, small tables have a higher variance in their achieved load factors. Therefore, improved load by the standard deviation is possible by just retrying a constant number of times in expectation.

(2) Median. For some configurations, small tables not only enable higher load factors because of the variance, but also because their median is higher than the load threshold. The effect is even more pronounced for ordinary binary cuckoo hashing, where a table with $M = 500$ has a median load factor of 56% (see Figure 2). This is significantly more than the asymptotic load threshold of 50%. A similar observation can be found in perfect hashing: The information theoretic minimal space to store an MPHF with small N is significantly lower than the asymptotic value for $N \to \infty$ [45].

(3) Space Usage. A metric for the lookup efficiency in irregular cuckoo hash tables is the average number of hash functions. For any desired average number of hash functions $d' \in \mathbb{R}$, the best load thresholds are given by a combination of $\lfloor d' \rfloor$ and $\lceil d' \rceil$ hash functions [10]. This picture changes fundamentally in the context of PHFs because the choice of hash functions is stored in binary coding. While, for example, an irregular cuckoo hash table with 50% 2-choice and 50% 4-choice has 3 choices on average, the storage space of the corresponding PHF is only $0.5\log(2) + 0.5\log(4) = 1.5 < \log(3)$. The configurations in Figure 1 all need the same storage space, but the median load factor increases the farther we are from the optimal configuration derived in Ref. [10].

Conclusion. Smaller cuckoo hash tables enable higher load factors than larger tables. Equivalently, by making the tables smaller, we can achieve the same load factor using a hash function mixture that needs less space. Even though we have to store a seed because the variance in the load factors is higher, we can use the described effects to save overall space.

Copyright © 2023
Copyright for this paper is retained by the authors

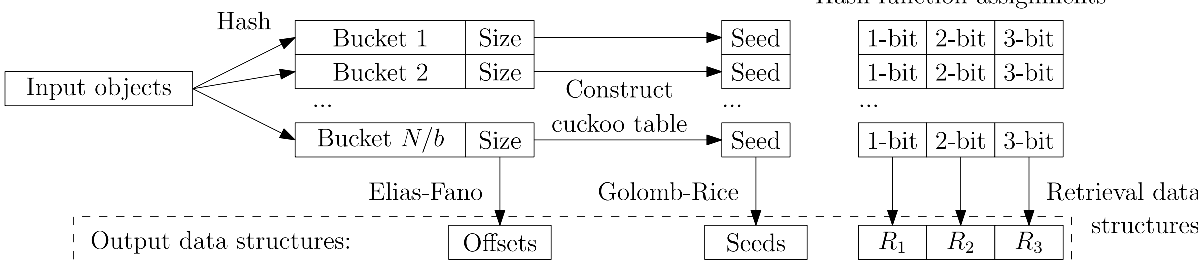

Figure 3: Data flow during construction. Objects are hashed to buckets of expected equal size. Within each bucket, a cuckoo hash table is constructed. The resulting hash function assignments from all small hash tables are stored together in three large retrieval data structures.

5 SicHash Perfect Hash Functions

We are now ready to introduce the main result of our paper: SicHash perfect hash functions. SicHash combines four results: Using cuckoo hashing for perfect hashing, making cuckoo hashing irregular, overloading small tables, and fast space efficient retrieval.

5.1 Construction. Building a SicHash function consists of the main steps partitioning, cuckoo hashing, storing bucket metadata, and constructing the retrieval data structures. Figure 3 gives an overview over the process.

Partitioning. First, we hash the input objects to a number of buckets that all have the same expected size b, for example $b = 5000$. The rather small size enables overloading and also keeps the storage space during construction small enough that the whole table fits into the cache.

Cuckoo Hashing. Within each bucket, we generate an irregular cuckoo hash table, using the same load factor as the overall perfect hash function. The number of cells in each of the small cuckoo hash tables is determined by the number of objects hashed to it, so the small tables have different sizes. However, the probability of a successful construction is similar for all of them. To determine how many hash functions (and therefore candidate cells) should be used for each object, we hash each object to a *class*. A certain percentage p_1 of objects is placed with $d = 2$ choices, a percentage p_2 with $d = 4$ choices, and the remaining objects with $d = 8$ choices.

To insert objects into the hash tables, we use *rattle kicking* [29] instead of the classical random walk. Rattle kicking maintains a counter for each object, indicating how often it was moved to a new cell. An object is then only evicted from its cell when its rattle counter is lower than the counter of the object to be inserted into the same cell. The next hash function index to use for inserting is the rattle counter modulo d. With rattle kicking, we can avoid the cost of random number generation and empirically need a lower number of steps during the insertion. For normal cuckoo hash tables, storing the counter would decrease the space efficiency. In our case, we only store the hash table temporarily, and also need the hash function index to construct the PHF later. This makes SicHash an attractive application for rattle kicking. Constructing a bucket may fail, in particular, when we configure a high degree of overloading. In this case, construction is retried while incrementing a seed value determining the used hash functions.

Storing Bucket Metadata. The result is a number of small hash tables, each with a seed leading to successful construction. In order to determine a global PHF, we need to offset each small table. We can do that by storing the exclusive prefix sum of table sizes using Elias-Fano coding. When trying out seeds, we can simply count up, starting with 0. Construction usually succeeds with one of the first few seeds, so we can compress the list of seeds using Golomb-Rice Codes. In practice, storing the two sequences in arrays offers better query time and negligible space overhead.

Retrieval. Now we only need to store the assignment from objects to cells within the small hash tables by storing which of the hash functions finally placed each object. Because the hash tables are irregular, we get indices of 1, 2, and 3 bits. While small hash tables enable overloading, retrieval data structures in contrast profit from handling many objects and can achieve overheads as low as 1% [13]. We therefore build 3 large retrieval data structures that hold the 1, 2, and 3-bit values from *all* the small hash tables. The space usage of the final PHF is dominated by the retrieval data structures.

Parameters. A SicHash PHF has three main tuning parameters: The load factor $\alpha = N/M$ to try construction for, and the class sizes for irregular cuckoo hashing, p_1 and p_2. Ignoring overloading, it is then possible to numerically determine a configuration that maximizes the load factor (see Section 7). Calculating efficient configurations *with* overloading remains an open problem.

Copyright © 2023
Copyright for this paper is retained by the authors

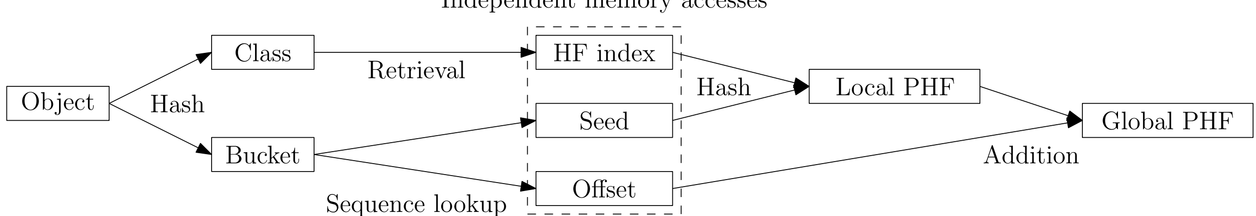

Figure 4: Data flow during a query. From the class of an object, we get the retrieval data structure to query. From the bucket index, we get the bucket's seed and offset. Combining this, we get a global hash function value.

When a specific space budget of β bits per object for the retrieval data structures is given, we get $p_2 = 3 - \beta - 2p_1$. The only remaining parameter p_1 can then be selected from the interval $[\max(0, 2-\beta), (3-\beta)/2]$.

5.2 Queries. A query for an object first hashes it to find its class, e.g., its number of candidate cells in the small hash table. This determines which retrieval data structure needs to be queried for the hash function index. Additionally, the object is hashed to find its bucket, and therefore the seed and offset. The value of the PHF is then given by hashing the object with the retrieved hash function index and the bucket's seed, and by adding the bucket's offset. Figure 4 gives an illustration for the data flow during a query, showing that all three memory access operations are independent of each other, so they can be prefetched or performed in parallel.

6 Enhancements

SicHash lends itself to numerous enhancements, which we outline below.

Minimal Perfect Hashing. SicHash can be converted to an MPHF by applying the same technique as in PTHash [38]. The idea is to re-map objects with hash function values $> N$ to smaller values by using an Elias-Fano coded sequence of size $M - N$. Refer to Section 3.2 for details. Note that the number of lower bits of the Elias-Fano coded sequence only depends on the load factor that was used before re-mapping. In practice, we can therefore use a compile-time parameter for faster bit operations in the Elias-Fano sequence.

Parallel Construction. Perfect hash functions can always be parallelized trivially by introducing a new layer on top of the data structure. SicHash can be parallelized more directly and without effect on the query speed. The small cuckoo hash tables of each bucket can be constructed embarrassingly parallel. Retrieval data structures can also be computed in parallel at some small space overhead linear in the number of processors and without query overhead [13].

External Memory Construction. SicHash can be adapted to very large inputs: First use external sorting to partition the objects into buckets. Then, for each bucket, find a cuckoo hash table – outputting sequences of seeds, offsets, and key-value pairs for the retrieval data structures. The latter can be fed into an external memory construction of the retrieval data structures [13].[4]

7 Analysis

THEOREM 7.1. *Parameters of SicHash can be chosen such that the expected construction time is linear in the input size N, that the query time is constant, and that the expected space consumption is* $O(N)$ *bits.*

Proof. (Outline) To be able to show linear construction time, we look at the case of constant expected bucket size, a bucket load factor that ensures constant success probability of cuckoo hash table construction, space efficient encoding of offsets using Elias-Fano coding, Golomb-Rice coding of seeds, and cuckoo table insertion using BFS. We also assume that we use a retrieval data structure that needs $O(N)$ bits, can be constructed in linear time and supports queries in constant time. After this proof, we informally discuss what changes for the simple implementation used in our experiments.

Query time. A query evaluates a constant number of hash functions (constant time), performs one access to a retrieval data structure (constant time as assumed above), and decodes a number each in an Elias-Fano coded and a Golomb-Rice coded sequence. Decoding the numbers boils down to constant time $select_1$-operations in a bit vector (see Section 2).

Space. The retrieval data structure uses a linear number of bits by our assumption. With average bucket size b, the Elias-Fano data structure takes $\left(2 + \log \frac{Mb}{N}\right) \frac{N}{b} = O(N)$ bits. For constant success probability of construction, the seed has a geometric distribution and constant expected length, i.e., expected

[4] BuRR [13] sorts elements by a hashed starting position in an equation system. By making this position monotonic in the bucket index one could save that sorting step.

Copyright © 2023
Copyright for this paper is retained by the authors

space consumption $\mathrm{O}\left(\frac{N}{b}\right) = \mathrm{O}(N)$ bits. When constructing an MPHF, the re-mapping (see Section 6) takes $(M-N)(2+\log\frac{N}{M-N}) = N\frac{1}{\alpha-1}(2+\log\frac{\alpha}{1-\alpha}) = \mathrm{O}(N)$ bits, where α is the load factor before compaction.

Construction Time. Constructing the retrieval data structures takes linear time by assumption. Building the data structures for seeds and offsets is obviously possible in linear time. Construction time for a bucket is at most quadratic in the bucket size (and, with constant success probability, retries contribute only a constant factor in expectation). With constant expected bucket size, we get the same execution time as a bucket sorting algorithm that uses a quadratic algorithm per bucket, which is expected linear [40, Theorem 5.9]. A similar argument can also be found in RecSplit [15]. □

Our implementation gains simplicity and query speed by storing offsets and seeds directly using $\log M$ bits. For this to be space efficient, we would need average bucket size $b = \Omega(\log N)$. At least with our simple estimation of construction time, this would result in superlinear construction time of $\Omega(N\log N)$. This can be improved using faster construction algorithms. For example, using the Hopcroft-Karp-Karzanov [24] algorithm, time $\mathrm{O}\left(N\sqrt{\log N}\right)$ can be proven. Assuming a result for random graph matching [1] also transfers to our case, we would even get $\mathrm{O}(N\log\log N)$. At least when we are sufficiently far from the load threshold, various previous results indicate that linear construction time is possible [17, 44, 21, 19, 26, 27]. With overloading, tight construction time bounds remain an open problem.

Load Thresholds. For a fixed configuration of fractions p_1, p_2, p_3, we now use existing theory to derive a threshold for the asymptotically maximum load factor $c^* = c^*(p_1,p_2,p_3)$ when the bucket size is large. The hope is that the threshold is also indicative of achievable loads for small to medium bucket sizes. We use a framework by Lelarge [32], which is quite challenging to understand and formally apply. Introducing the underlying ideas of belief propagation and local weak convergence, let alone carrying out the required technical work for our case is beyond the scope of this paper. What we can do is restate the equations characterizing the load threshold c_d^* when each object has the same number d of hash functions and then point out how these equations have to be adapted for the irregular case. The following equations are taken from Ref. [32, Theorems 1.1 and 4.1], specialised for $(h,k,\ell) = (d,1,1)$ and simplified.

$$\begin{aligned}
F(q,c) &= 1-(1-g^B(q))^d + \tfrac{1-\mathrm{e}^{-cdq}(1+cdq)}{c}\\
g^B(q) &= \mathrm{e}^{-cdq}, \qquad g^A(p) = (1-p)^{d-1}\\
X(c) &= \{F(q,c) \mid q\in[0,1], g^A(g^B(q)) = q\}\\
c_d^* &= \sup\{c>0 \mid \inf X(c) = 1\}.
\end{aligned}$$

Modified for our case, when objects use $d_1{=}2$, $d_2{=}4$, $d_3{=}8$ hash functions with probability p_1, p_2, p_3, respectively, not much changes in the terms that handle belief propagation at the table cells, except that d has to be replaced with $\bar{d} := \sum_i p_i d_i$. From the point of view of an object, three possibilities have to be taken into account, which occur with probabilities p_1, p_2, p_3 if the object is chosen uniformly at random, and with probabilities $p_1d_1/\bar{d}$, $p_2d_2/\bar{d}$, $p_3d_3/\bar{d}$ if the object is chosen with a probability proportional to the number of table cells it is incident to. The correspondingly modified equations read as follows.

$$\begin{aligned}
F(q,c) &= 1-\sum_i p_i(1-g^B(q))^{d_i} + \tfrac{1-\mathrm{e}^{-c\bar{d}q}(1+c\bar{d}q)}{c}\\
g^B(q) &= \mathrm{e}^{-c\bar{d}q}, \qquad g^A(p) = \sum_i \frac{p_i d_i}{\bar{d}}(1-p)^{d_i-1}\\
X(c) &= \{F(q,c) \mid q\in[0,1], g^A(g^B(q)) = q\}\\
c^* &= c^*(p_1,p_2,p_3) = \sup\{c>0 \mid \inf X(c) = 1\}.
\end{aligned}$$

Consider solutions (q,c) to $g^A(g^B(q)) = q$. For any c, a trivial solution is $(0,c)$. Any non-trivial solution is uniquely determined by $\lambda := c\bar{d}q$ since we can compute $q(\lambda) = g^A(g^B(q)) = g^A(e^{-\lambda})$ and $c(\lambda) = \frac{\lambda}{q(\lambda)\bar{d}}$ from it. The corresponding value $F(\lambda) = F(q(\lambda),c(\lambda))$ is

$$F(\lambda) = 1-\sum_i p_i(1-\mathrm{e}^{-\lambda})^{d_i} + \frac{g^A(e^{-\lambda})\bar{d}}{\lambda}(1-\mathrm{e}^{-\lambda}(1+\lambda))$$

We can then rewrite $c^* = \sup\{c>0 \mid \nexists\lambda > 0 : F(\lambda) < 1 \text{ and } c = c(\lambda)\}$ and obtain numerical approximations of c^* by plotting $F(\lambda)$ and $c(\lambda)$. Indeed, it seems that $c^* = c(\lambda^*)$ for the largest root λ^* of $F(\lambda)-1$. Needless to say, a lot more work would be required to rigorously defend this method.

8 Experiments

The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License: https://github.com/ByteHamster/SicHash. The code for the comparison with competitors is available on GitHub as well: https://github.com/ByteHamster/MPHF-Experiments. Both repositories also contain Docker containers that can build and run a simplified version of the experiments in about 90 minutes.

Experimental Setup. We run our experiments on an Intel Xeon E5-2670 v3 with a base clock speed of 2.3 GHz. The L1, L2, and L3 caches have a size of 768 KiB, 6 MiB, and 60 MiB, respectively. The machine runs Ubuntu 20.04 with Linux 5.10.0. We use the GNU

Copyright © 2023
Copyright for this paper is retained by the authors

Table 1: Selected configuration parameters from the Pareto front (see Figure 8). Even though RecSplit is minimal perfect, it is competitive with methods achieving N/M=0.95, so it is included in that section as well.

	B/Obj	Method	MObj/s	ns/Obj
N/M=1.0	2.01	RecSplit, l=5 b=100	2.85	350
	2.06	PTHash, lf=0.95, c=3.0	0.30	3333
	2.04	lf=0.97, p_1=45%, p_2=31%	2.94	340
	2.50	RecSplit, l=3, b=50	3.86	259
	2.51	PTHash, lf=0.95, c=6.0	3.06	327
	2.50	lf=0.9, p_1=21%, p_2=78%	4.91	204
N/M=0.95	1.78	RecSplit, l=8, b=250	0.76	1316
	1.76	PTHash, c=3.0	0.61	1639
	1.75	p_1=44%, p_2=41%	3.27	305
	2.09	RecSplit, l=5, b=50	3.17	315
	2.06	PTHash, c=5.8	2.67	375
	2.04	p_1=43%, p_2=18%	4.55	220
N/M=0.85	1.61	PTHash, c=4.6	2.85	351
	1.61	p_1=48%, p_2=51%	4.74	211
	1.91	PTHash, c=7.4	4.25	235
	1.89	p_1=45%, p_2=31%	5.30	189

C++ compiler version 11.1.0 with optimization flags `-O3 -march=native`. We also achieve comparable results on an AMD Ryzen 3950X with a base clock speed of 3.5 GHz. After small adaptions of the competitors to replace inline assembly and intrinsics, we even achieve comparable results on an ARM Neoverse-N1. Refer to Figure 9 for measurements on the two additional machines. As a retrieval data structure, we use Bumped Ribbon Retrieval [13], with two alternative configurations: $w = 64$ with 2-bit bumping info, and $w = 32$ with 1-bit bumping info. Instead of encoding the per-bucket metadata using Elias-Fano coding and Golomb-Rice coding, we use a plain array, which is faster to access and causes only a small space overhead for sufficiently large buckets. The extension to minimal perfect hashing stores the re-mapping with Elias-Fano coding, which is based on sdsl's [22] arrays of flexible bit width and the select data structures by Kurpicz [28]. For all our experiments, construction and queries are executed on a single thread. Objects are strings of uniform random length $\in [10, 50]$ containing random characters except for the zero byte. Given that SicHash and its competitors first hash each input object using an ordinary hash function, the evaluation is independent of the distribution of input objects.

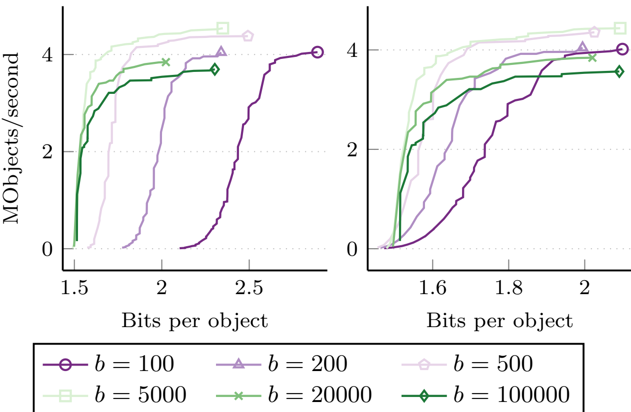

Figure 5: Pareto plot over the construction throughput of different SicHash configurations by bucket size. Load factor $N/M = 0.9$. Left: measured space usage. Right: hypothetical space usage, assuming that bucket metadata is encoded with Elias-Fano and Golomb-Rice codes.

8.1 SicHash Configurations. The query performance is independent of the choice of p_1 and p_2, so we focus the comparison on the construction performance.

Cuckoo Placement. It is possible to construct the cuckoo hash table with random walk insertion, as well as matching based methods. In our experiments, the random walk variant *rattle kicking* [29] is usually faster than a construction based on Hopcroft-Karp-Karzanov [24].

Bucket Size. For larger buckets, the relative overhead of encoding the metadata is reduced, but they can be overloaded less. Figure 5 (left) shows a Pareto front[5] of SicHash configurations using different bucket sizes and indicates that a bucket size of $b = 5000$ is optimal, which is why we choose that parameter in all other measurements. Figure 5 (right) shows hypothetical values for the space usage when assuming that the per-bucket metadata is encoded with Elias-Fano and Golomb-Rice coding instead of arrays (see Section 5).

Parameter Choices. A guideline on how to choose parameters for an efficient PHF construction can be determined by running a benchmark of multiple configurations and determining the Pareto front. Table 1 gives exemplary configuration parameters p_1, p_2 that are on the Pareto front on our machine, as well as a small selection of competitors achieving the same storage space. As discussed in Section 3.1, an ordinary cuckoo hash table leads to a PHF with a load factor of 97.68% and 2 bits per object. Our implementation of SicHash achieves the same

[5] A configuration is on the Pareto front if it is not dominated by any other configuration with respect to both construction time and space consumption.

Copyright © 2023
Copyright for this paper is retained by the authors

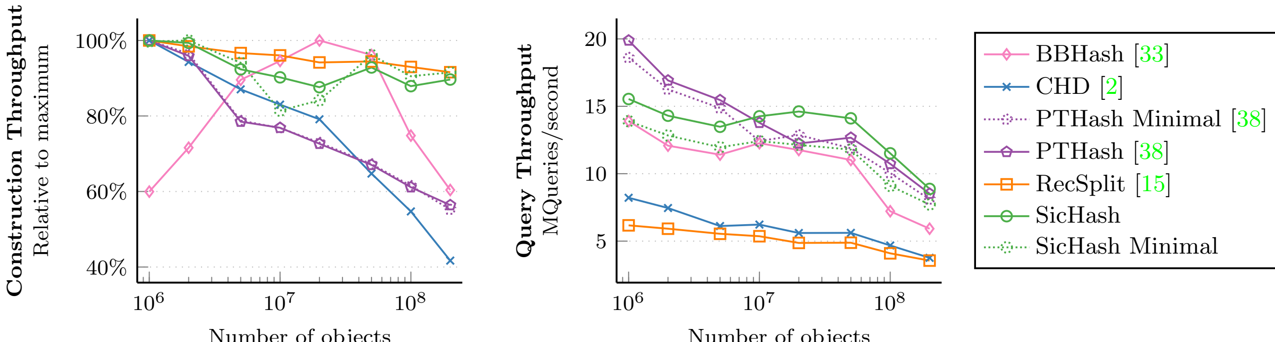

Figure 6: Comparison of different competitors by number of objects N. Left: Construction throughput relative to each method's maximum. Right: Query throughput. For the PHFs, the load factor is 0.95. The parameters are selected from the Pareto front in a way that all PHFs achieve a space usage of 1.8 bits per object, BBHash achieves 3.5 bits per object, and all other MPHFs achieve a space usage of 2.2 bits per object.

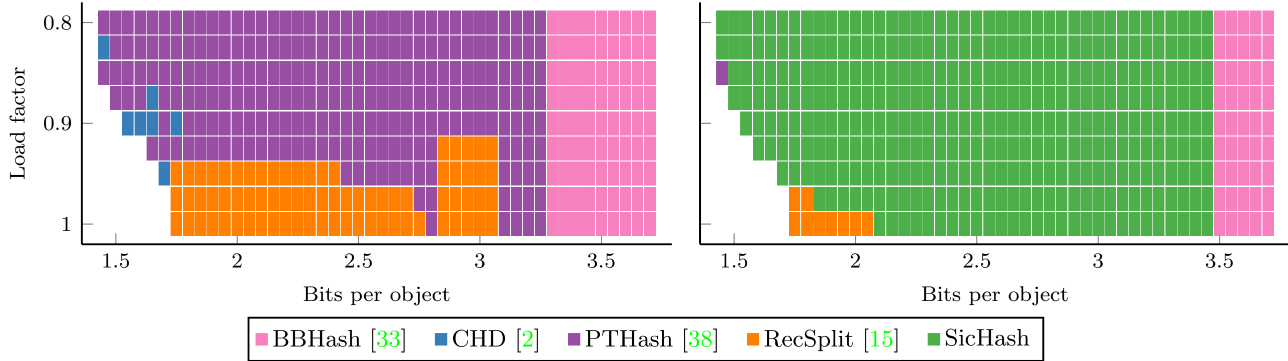

Figure 7: Heat map showing which PHF construction method has the fastest construction time for each given combination of load factor and space usage. Left: Algorithms without SicHash. Right: Algorithms including SicHash. SicHash covers most of the configurations.

load factor using only 1.84 bits per object ($p_1 = 49\%$, $p_2 = 22\%$). When constructing without overloading, placing the objects in the cuckoo hash tables takes about 50% of the time and constructing the retrieval data structure also takes about 50% of the time. When increasing the amount of overloading, placing objects in the cuckoo hash table takes longer because more retries are needed.

8.2 Comparison with Competitors. For demonstrating the performance of SicHash, we compare it to competitors from the literature (see Section 3.2). From the cmph library, we include CHD [2], BDZ [5], BMZ[6] [3], and FCH[6] [20]. Additionally, we include PTHash [38], RecSplit[6] [15], and BBHash[6] [33].

[6]Method only supports construction of MPHF, not PHF. Included in plots for all load factors.

Construction Scaling. Figure 6 (left) shows how the construction throughput scales with N. The configuration parameters of each method are selected from the Pareto front in a way that all PHFs with a load factor of 0.95 need 1.8 bits per object, BBHash needs 4.0 bits, and all other MPHFs need 2.2 bits per object. To be more fair about the different load factors and space used, the construction throughput is given *relative* to each competitor's maximum performance. The construction throughput of PTHash, BBHash and CHD drops significantly when increasing N, while SicHash and RecSplit stay more constant. This can be explained by the fact that SicHash and RecSplit hash objects to buckets of expected constant size that are then constructed independently. The other competitors, in contrast, access an array with size proportional to N, which leads to lower cache locality. While putting an additional layer on top of the data structures for more cache locality is always

Copyright © 2023
Copyright for this paper is retained by the authors

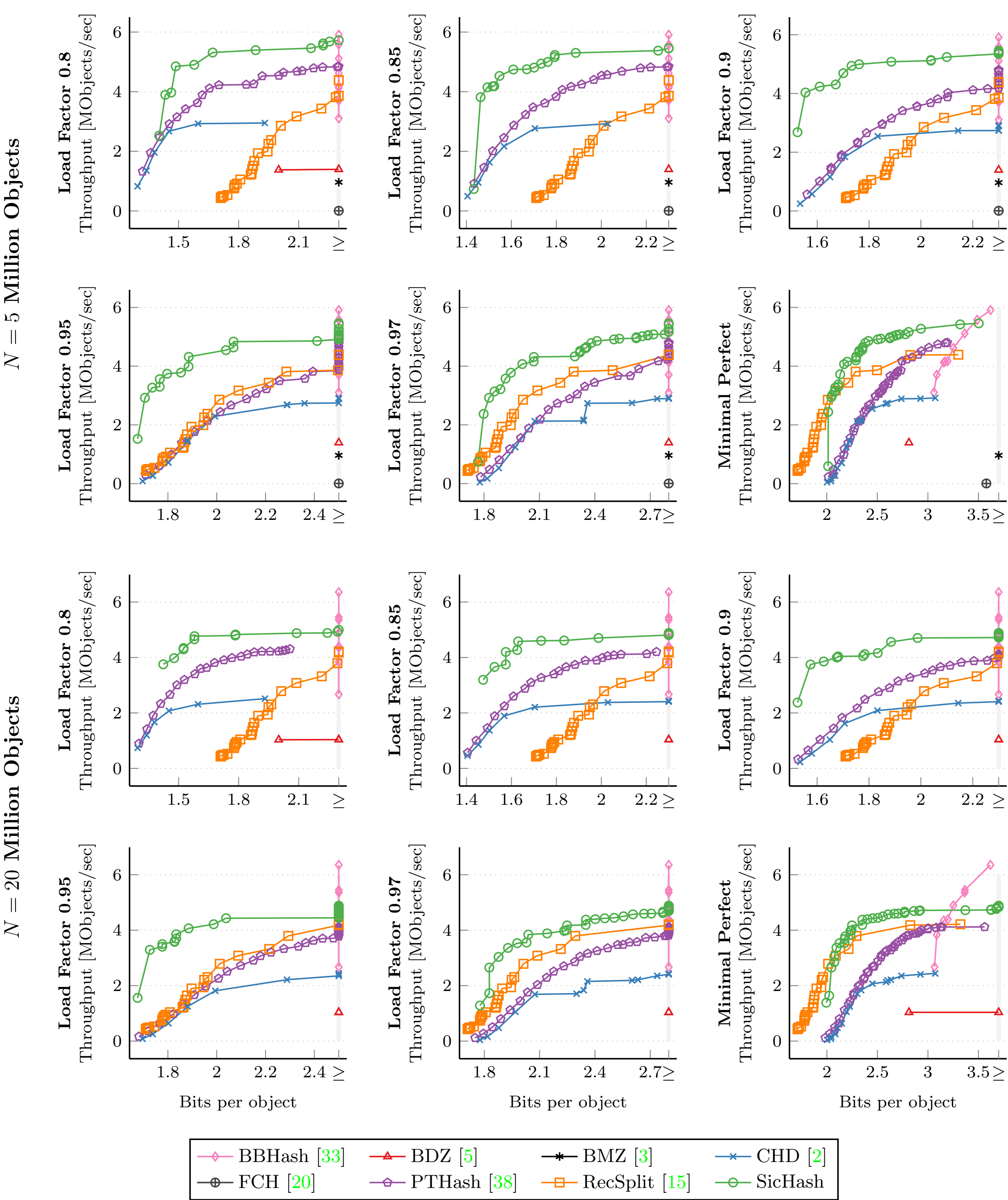

Figure 8: Space usage vs construction throughput for different load factors. Each line is a Pareto front showing configurations of a method that are not dominated by other configurations. Each plot contains a PHF that achieves *at least* the specific load factor. In particular, minimal perfect hash functions are included in all plots. The top six plots use $N = 5$ Million objects, while the bottom six plots use $N = 20$ Million objects. Methods that are entirely uncompetitive are excluded from the plot with $N = 20$ Million objects.

Copyright © 2023
Copyright for this paper is retained by the authors

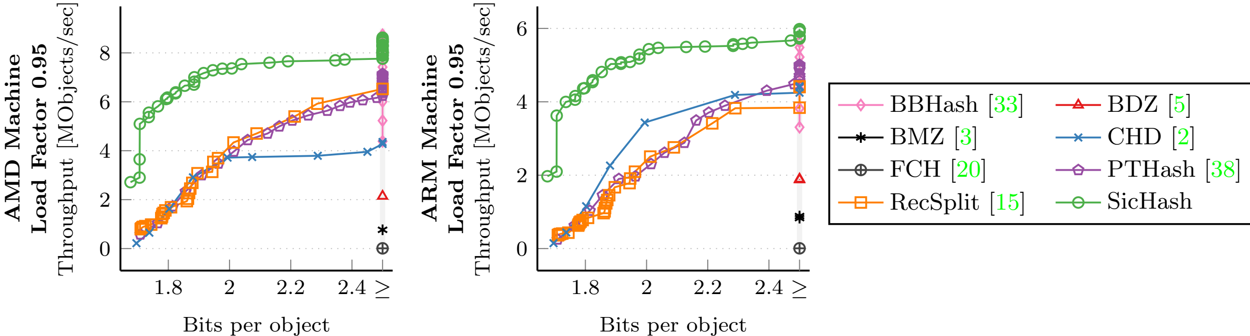

Figure 9: Space usage vs construction throughput on two additional machines. Uses $N = 5$ Million objects.

possible, this would affect query time and space usage. The initial increase of BBHash's construction throughput can be explained by the constant startup overhead of launching a thread even in single-threaded mode.

Query Scaling. The query throughput in Figure 6 drops when increasing N because the data structure gets larger and can be cached less. One of the main design goals of PTHash is its fast queries. SicHash comes close to PTHash or even surpasses it in terms of query throughput, while having a more favorable trade-off between space usage and construction performance. The minimal perfect variant has a query time overhead of about 5–10%.

Trade-Off. For each competitor, we run experiments with a number of configurations to provide a trade-off between construction time and space usage. Figure 8 gives Pareto fronts of our method and competitors for different load factors and input sizes N. SicHash dominates the Pareto front for load factors from 0.85 to 0.97. While there are better competitors for very space efficient or very large data structures, SicHash covers a wide range of configurations (see also Figure 7). As visible in Table 1, SicHash can be constructed up to 4.3 times faster than the next best competitor. For MPHFs, RecSplit produces the smallest data structures, but SicHash is competitive for a range of configurations while having significantly faster queries (see Figure 6). Measurements on two additional hardware architectures give similar results (see Figure 9).

9 Conclusion and Future Work

With SicHash, we present a new perfect hash function which places objects in a number of small, irregular cuckoo hash tables. Making the tables small enables *overloading*, which achieves higher load factors than the asymptotic bound. Using irregular cuckoo hashing enables fine-grained control over the load factors and lower space usage. We then use space efficient retrieval data structures to store the final placement. Our implementation improves the state of the art in perfect hash functions for a wide range of load factors and space usage configurations. It profits from the fact that building cuckoo hash tables is a more directed approach for finding bijections than the brute force methods used at the core of many competitors.

Future work might include a parallel construction algorithm and construction using a different insertion strategy than rattle kicking. From a theoretical point, it would be interesting to look at construction success probabilities for overloaded cuckoo hash tables.

Acknowledgements. The authors would like to thank Martin Dietzfelbinger for early discussions leading to this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

Copyright © 2023
Copyright for this paper is retained by the authors

References

[1] Hannah Bast, Kurt Mehlhorn, Guido Schäfer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory Comput. Syst.*, 39(1):3–14, 2006. doi:10.1007/S00224-005-1254-Y.

[2] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009. doi:10.1007/978-3-642-04128-0_61.

[3] Fabiano C Botelho, David M Gomes, and Nivio Ziviani. A new algorithm for constructing minimal perfect hash functions. *differences*, 100(2):09, 2004.

[4] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007. doi:10.1007/978-3-540-73951-7_13.

[5] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013. doi:10.1016/J.IS.2012.06.002.

[6] Marshall D. Brain and Alan L. Tharp. Perfect hashing using sparse matrix packing. *Inf. Syst.*, 15(3):281–290, 1990. doi:10.1016/0306-4379(90)90001-6.

[7] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. Weighted bloom filter. In *ISIT*, pages 2304–2308. IEEE, 2006. doi:10.1109/ISIT.2006.261978.

[8] Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. Meraculous: De novo genome assembly with short paired-end reads. *PLOS ONE*, 6(8):1–13, 08 2011. doi:10.1371/journal.pone.0023501.

[9] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Inf. Process. Lett.*, 43(5):257–264, 1992. doi:10.1016/0020-0190(92)90220-P.

[10] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *ICALP (1)*, volume 6198 of *Lecture Notes in Computer Science*, pages 213–225. Springer, 2010. doi:10.1007/978-3-642-14165-2_19.

[11] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In *ESA*, volume 6942 of *Lecture Notes in Computer Science*, pages 615–627. Springer, 2011. doi:10.1007/978-3-642-23719-5_52.

[12] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2005. doi:10.1007/11523468_14.

[13] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SEA.2022.4.

[14] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.

[15] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *ALENEX*, pages 175–185. SIAM, 2020. doi:10.1137/1.9781611976007.14.

[16] Robert Mario Fano. On the number of bits required to implement an associative memory. Technical report, MIT, Computer Structures Group, 1971. Project MAC, Memorandum 61".

[17] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/S00224-004-1195-X.

[18] Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Struct. Algorithms*, 41(3):306–333, 2012. doi:10.1002/RSA.20426.

[19] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM J. Comput.*, 42(6):2156–2181, 2013. doi:10.1137/100797503.

[20] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *SIGIR*, pages 266–273. ACM, 1992. doi:10.1145/133160.133209.

[21] Alan M. Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. In *APPROX-RANDOM*, volume 5687 of *Lecture Notes in Computer Science*, pages 490–503. Springer, 2009. doi:10.1007/978-3-642-03685-9_37.

[22] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

[23] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966. doi:10.1109/TIT.1966.1053907.

[24] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[25] Gerhard Jaeschke. Reciprocal hashing: A method for generating minimal perfect hashing functions. *Commun. ACM*, 24(12):829–833, 1981. doi:10.1145/358800.358806.

[26] Megha Khosla. Balls into bins made faster. In *ESA*, volume 8125 of *Lecture Notes in Computer Science*, pages 601–612. Springer, 2013. doi:10.1007/978-3-642-40450-4_51.

[27] Megha Khosla and Avishek Anand. A faster algorithm for cuckoo insertion and bipartite matching in large graphs. *Algorithmica*, 81(9):3707–3724, 2019. doi:10.1007/S00453-019-00595-4.

Copyright © 2023
Copyright for this paper is retained by the authors

[28] Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

[29] William Kuszmaul. Fast concurrent cuckoo kick-out eviction schemes for high-density tables. *CoRR*, abs/1605.05236, 2016. doi:10.48550/arXiv.1605.05236.

[30] Reinhard Kutzelnigg. An improved version of cuckoo hashing: Average case analysis of construction cost and search operations. In *IWOCA*, pages 253–266. College Publications, 2008.

[31] Eric Lehman and Rina Panigrahy. 3.5-way cuckoo hashing for the price of 2-and-a-bit. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 671–681. Springer, 2009. doi:10.1007/978-3-642-04128-0_60.

[32] Marc Lelarge. A new approach to the orientation of random hypergraphs. In *SODA*, pages 251–264. SIAM, 2012. doi:10.1137/1.9781611973099.23.

[33] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *SEA*, volume 75 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.SEA.2017.25.

[34] Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Trans. Inf. Theory*, 47(2):569–584, 2001. doi:10.1109/18.910575.

[35] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2014. doi:10.1007/978-3-319-07959-2_12.

[36] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[37] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

[38] Giulio E. Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. doi:10.1145/3404835.3462849.

[39] Robert F. Rice. Some practical universal noiseless coding techniques. *Jet Propulsion Laboratory, JPL Publication*, 1979.

[40] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.

[41] Renzo Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Commun. ACM*, 20(11):841–850, 1977. doi:10.1145/359863.359887.

[42] Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. In *ICALP*, volume 107 of *LIPIcs*, pages 102:1–102:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.ICALP.2018.102.

[43] Stefan Walzer. Peeling close to the orientability threshold - spatial coupling in hashing-based data structures. In *SODA*, pages 2194–2211. SIAM, 2021. doi:10.1137/1.9781611976465.131.

[44] Stefan Walzer. Insertion time of random walk cuckoo hashing below the peeling threshold. In *ESA*, volume 244 of *LIPIcs*, pages 87:1–87:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ESA.2022.87.

[45] Sean A. Weaver and Marijn Heule. Constructing minimal perfect hash functions using SAT technology. In *AAAI*, pages 1668–1675. AAAI Press, 2020.

Copyright © 2023
Copyright for this paper is retained by the authors