

# Bipartite ShockHash: Pruning ShockHash Search for Efficient Perfect Hashing

Hans-Peter Lehmann  

Karlsruhe Institute of Technology, Germany

Peter Sanders  

Karlsruhe Institute of Technology, Germany

Stefan Walzer  

Karlsruhe Institute of Technology, Germany

---

## Abstract

A minimal perfect hash function (MPHF) maps a set of  $n$  keys to the first  $n$  integers without collisions. Representing this bijection needs at least  $\log_2(e) \approx 1.443$  bits per key, and there is a wide range of practical implementations achieving about 2 bits per key. Minimal perfect hashing is a key ingredient in many compact data structures such as updatable retrieval data structures and approximate membership data structures.

A simple implementation reaching the space lower bound is to sample random hash functions using brute-force, which needs about  $e^n \approx 2.718^n$  tries in expectation. ShockHash recently reduced that to about  $(e/2)^n \approx 1.359^n$  tries in expectation by sampling random graphs. With bipartite ShockHash, we now sample random *bipartite* graphs. In this paper, we describe the general algorithmic ideas of bipartite ShockHash and give an experimental evaluation. The key insight is that we can try all combinations of two hash functions, each mapping into one half of the output range. This reduces the number of sampled hash functions to only about  $(\sqrt{e/2})^n \approx 1.166^n$  in expectation. In itself, this does not reduce the asymptotic running time much because all combinations still need to be tested. However, by filtering out hash function candidates that do not cover all output values *before* combining them, we can reduce this to less than  $1.175^n$  combinations in expectation.

Our implementation of bipartite ShockHash is up to 3 orders of magnitude faster than original ShockHash. Inside the RecSplit framework, bipartite ShockHash-RS enables significantly larger base cases, leading to a construction that is, depending on the allotted space budget, up to 20 times faster. In our most extreme configuration, ShockHash-RS can build an MPHF for 10 million keys with 1.489 bits per key (within 3.3% of the lower bound) in about half an hour, pushing the limits of what is possible.

**2012 ACM Subject Classification** Theory of computation → Data compression; Information systems → Point lookups

**Keywords and phrases** compressed data structure, perfect hashing, bit parallelism, vector instructions

**Supplementary Material** *Software (Source code):* <https://github.com/ByteHamster/ShockHash>  
*Software (Comparison with competitors):* <https://github.com/ByteHamster/MPHF-Experiments>

**Funding** 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).



## 1 Introduction

A perfect hash function (PHF) is a collision-free mapping from a set  $S$  of  $n$  keys to the first  $m$  integers. If  $m = n$ , the function is called a *minimal* perfect hash function (MPHF) and forms a bijection between  $S$  and  $[n]$ . Minimal perfect hashing is a key ingredient in many compact data structures. For example, it can be used to implement hash tables with guaranteed constant access time [10], updatable retrieval data structures [14], and approximate membership data structures [3, 8]. The space lower bound is  $n \log_2 e$  bits and can be achieved by a simple brute-force technique which samples  $e^n \approx 2.718^n$  random hash functions in expectation and stores the index of the first function that is minimal perfect.

The recently introduced ShockHash [11] is a fast approach to minimal perfect hashing. It samples a pair of hash functions and tries to find a mapping such that each input key is assigned to one of its two candidate positions. In graph terminology, ShockHash samples random graphs until one of them can be oriented such that each node has indegree 1. The paper shows that in expectation, about  $(e/2)^n \approx 1.359^n$  graphs need to be sampled, so storing the hash function seed needs about  $n \log_2(e) - n$  bits. ShockHash then stores a 1-bit retrieval data structure, mapping each key to the choice between its two candidate positions, taking about  $n$  bits.

This paper introduces two new ideas. First, instead of using a pair of fresh hash functions for each construction attempt we build a growing pool of hash functions and consider all pairs that can be formed from this pool. This reduces the number of hash function evaluations exponentially. Second, we let the two hash functions hash to disjoint ranges, meaning we effectively sample a bipartite graph where each edge has one endpoint in both partitions. In this bipartite setting, the hash functions of both partitions need to be individually surjective. We can therefore filter the set of candidate hash functions in each partition individually – before testing all combinations. This improves the construction performance by an exponential factor, requiring less than  $1.175^n$  hash function tries. In this paper, we describe basic algorithmic ideas and give an experimental evaluation.

RecSplit [7] is a very space-efficient framework for perfect hash functions, which originally uses the brute-force technique as its base case. The construction performance can be improved significantly by plugging ShockHash into the base case of RecSplit. Bipartite ShockHash improves this by a factor of 20 again. For 10 million keys, RecSplit achieves a space usage of 1.584 bits per key within half an hour. In a similar amount of time, 1.499 bits per key can be achieved by massive parallelization of the approach using a GPU [4]. ShockHash [11] achieves a space usage of 1.523 bits per key using a single CPU thread. Now, bipartite ShockHash achieves a new record of just 1.489 bits per key, while using just a single CPU thread.

**Outline.** In Section 2, we describe RecSplit and plain ShockHash which form the basis of our paper. We then briefly review other related work in Section 3. In Section 4 we explain bipartite ShockHash including variants and refinements. We conduct detailed experiments in Section 5. Finally, we give a conclusion in Section 6.

**Our Contributions.** With *bipartite* ShockHash, we present a significant improvement of the ShockHash perfect hash function that has exponentially faster construction time. When plugged into the RecSplit framework, our implementation is about 20 times faster than the original and is the first perfect hash function to achieve 1.489 bits per key for 10 million keys.

## 2 Preliminaries

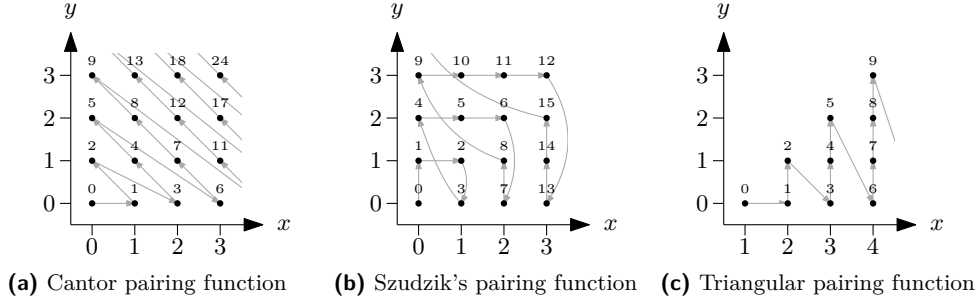
In the following section, we explain ShockHash [11] and RecSplit [7], the methods that our paper is based on. We also give a short explanation of retrieval data structures and pairing functions.

**Retrieval Data Structures.** A retrieval data structure (or *static function* data structure) stores a function  $S \rightarrow \{0, 1\}^r$  that maps each key of the input set  $S$  to a specific  $r$ -bit value. Because it may return arbitrary values for keys not in  $S$ , it is possible to represent the function without representing  $S$  itself. In particular, for  $r = 1$ , BuRR [6] reduces function evaluation to XORing a hash function value with a segment of a precomputed table and reporting the parity of the result. This table can be determined by solving a nearly diagonal system of linear equations. In practice, BuRR needs about  $1.007n$  bits and can be constructed in linear time.

**ShockHash.** ShockHash [11] samples a pair of hash functions and tries to find a mapping such that each key is assigned to one of its two candidate positions. In graph terminology, ShockHash samples random graphs until one of them can be *oriented* such that each node has indegree 1. The paper shows that each way to pick an assignment from an edge to one of its two incident vertices essentially provides a new chance for finding a perfect hash function. An important ingredient for this is to show that the number of orientations of such a graph is small. Because the graph can be checked for orientability in linear time, ShockHash can basically check  $2^n$  different hash functions in linear time, leading to exponential speedups. The paper shows that in expectation, the hash function seed needs about  $n \log_2(e) - n$  bits. ShockHash then stores a 1-bit mapping indicating which candidate position was used for each key using a retrieval data structure, taking about  $n$  bits.

**RecSplit.** RecSplit [7] revolutionized the field of minimal perfect hashing close to the theoretical bound. It is the first method that achieves very small space usage in practice. As an initial step, RecSplit hashes  $N$  input keys to buckets of expected size  $b$ , where a typical value is  $b = 2000$ . Within each bucket, RecSplit then recursively *splits* the key set to smaller sets by trying hash functions using brute-force until one divides the keys in a specific way. This forms a tree, where the leaves are small sets of fixed size  $n$  (except the last leaf), where a typical value is  $n = 16$ . Within these leaves, RecSplit then calculates bijections by performing brute-force search for an MPH. RecSplit can be parallelized using multi-threading and using the GPU [4]. The GpuRecSplit paper also introduces *rotation fitting*, a technique that reduces the number of hash function evaluations by a factor of  $n$ . The idea of rotation fitting is to divide the keys into two sets and then cyclically rotate (modulo  $n$ ) all hash values of one of the sets. Both plain ShockHash and bipartite ShockHash can be used as a base case inside the RecSplit framework by replacing the brute force search in the leaves. This makes it possible to increase the leaf size and therefore increase the space efficiency.

**Pairing Functions.** A pairing function encodes two natural numbers in a single natural number. More precisely, a pairing function is a bijection between the grid  $\mathbb{N}_0^2$  and  $\mathbb{N}_0$ . We are interested in pairing functions that can be inverted efficiently. The most popular pairing function is the Cantor pairing function, which enumerates the 2D grid diagonally (see Figure 1a). It can be calculated by  $\text{pair}_c(x, y) = (x + y)(x + y + 1)/2 + y$ . Another pairing function is the one by Szudzik [16], which enumerates the 2D grid following the edges of a



■ **Figure 1** Illustrations of different pairing functions.

square (see Figure 1b). The pairing function can be calculated by  $\text{pair}_s(x, y) = y^2 + x$  if  $x < y$  and  $\text{pair}_s(x, y) = x^2 + x + y$  otherwise.

In this paper, we require a function that enumerates only those coordinates of the 2D grid with  $x > y$ . We will still call it a pairing function in slight abuse of traditional terminology. Our *triangular pairing function* (see Figure 1c) can be calculated by  $\text{pair}_t(x, y) = x(x-1)/2 + y$  with the intuition stemming from the *Little Gauss* formula. The basic idea for inverting our function  $(x', y') = \text{pair}_t^{-1}(z)$  is to set  $y = 0$  in the definition and solve for  $x$ . This gives  $x' = \lfloor 1/2 + \sqrt{1/4 + 2z} \rfloor$  and  $y' = z - \text{pair}_t(x', 0)$ . In our bipartite implementation, we use both our triangular pairing function and Szudzik's pairing function, depending on the distribution of the numbers we want to encode.

While pairing uses only integer operations, all three pairing functions rely on the square root operation and rounding for inverting. This means that inverting the functions in practice can lead to problems due to floating point inaccuracies. Whether inverting  $z$  succeeded can easily be checked by verifying that  $\text{pair}(x', y') = z$ . In our implementation, we check invertibility at construction time, so we do not get a run-time overhead during queries.

### 3 Related Work

In addition to RecSplit [7] and ShockHash [11] which we explained in the previous section, there is a wide range of perfect hash function constructions. The methods all have a different focus – while RecSplit is focused more on achieving very small space usage, others focus more on query speed. We only describe other approaches briefly here and refer to Ref. [12] for a more detailed overview of related work.

(M)PHFs with *Hash-and-Displace* [1, 9, 15] allow fast queries and asymptotically optimal space consumption. Each key  $x$  is first hashed to a small bucket  $b(x)$  of keys. For each bucket  $b$ , an index  $i(b)$  of a hash function  $f_{i(b)}$  is stored such that  $x \mapsto f_{i(b(x))}(x)$  is an injective function. For a particular bucket, this index is then found by trying hash functions using brute-force.

Perfect hashing through *fingerprinting* [2, 5, 13, 14] hashes the  $n$  keys to  $\gamma n$  positions using an ordinary hash function, where  $\gamma > 1$  is a tuning parameter. A bit vector of length  $\gamma n$  indicates positions to which exactly one key was mapped. Keys that caused collisions are handled recursively in another layer of the same data structure.

SicHash [12] uses a mix of 1–3 bit retrieval data structures to store hash function choices for each key. Compared to ShockHash, it does not achieve minimality directly and works best for about 2–3 bits per key.



```

Function construct( $S$ )
  surjectiveCandidates  $\leftarrow \emptyset$ 
  for  $s_0 = 0$  to  $\infty$ 
    if  $h_{s_0}$  is surjective on  $S$ 
      for  $s_1 \in$  surjectiveCandidates
        if  $\exists f \in \{0, 1\}^S : x \mapsto \frac{n}{2} \cdot f(x) + h_{s_{f(x)}}(x)$  is a bijection
          return  $f$  as retrieval data structure,  $\text{pair}_t(s_0, s_1)$ 
        surjectiveCandidates  $\leftarrow$  surjectiveCandidates  $\cup \{s_0\}$ 
Function evaluate( $x$ )
  return  $\frac{n}{2} \cdot f(x) + h_{s_{f(x)}}(x)$ 

```

■ **Figure 3** Pseudocode of bipartite ShockHash.

We linearly check seeds until we find a seed  $s_0$  that is surjective. Given that new candidate, we combine it with all previous candidates  $s_1$  by checking if the graph defined by the two hash functions  $h_0 : x \mapsto h_{s_0}(x)$  and  $h_1 : x \mapsto n/2 + h_{s_1}(x)$  is orientable. If it is orientable (meaning that we find a collision-free assignment from each key to one of the candidate hash values), we have found a perfect hash function. We only need to store the assignment from keys to their candidate hash function ( $h_0$  or  $h_1$ ) in a retrieval data structure and the two seeds  $s_0$  and  $s_1$ . If the combination with none of the previous seed candidates leads to an orientable graph, we add the new candidate  $s_0$  to the set of surjective candidates and search for the next one. Figure 3 gives a pseudocode for this algorithm.

Note that it does not matter which function we use for which partition of the graph. Switching the partitions just gives an isomorphic graph and does not influence orientability. We therefore always use the newly determined candidate directly and shift the old candidate by  $n/2$ . Also, we neglect the possibility that a hash function combined with itself on both partitions leads to successful construction.<sup>1</sup> This allows us to store the two seeds  $s_0$  and  $s_1$ , knowing that  $s_1 < s_0$ . We do so in one integer using our triangular pairing function that we explain in Section 2. Note that the pairing function enumerates the seed pairs in exactly the same order that we test them in. Compared to storing two variable-length integers, pairing reduces constant overheads.

## 4.1 Seed Candidate Generation

Until now, we have tested hash function seeds linearly, using a new hash function in each iteration. However, with this technique, hash function evaluations are still a bottle-neck in practice. In the following, we describe additional ways of coming up with a stream of surjective hash function candidates that we can then combine with previous candidates.

### 4.1.1 Rotation Fitting

Rotation fitting [4] creates new hash function candidates by dividing the key set into two subsets using a constant 1-bit hash function. Note that the subsets usually do not have the same size. Both sets are then hashed with the same hash function. Seed candidates are generated by cyclically rotating (mod  $n/2$ ) one of the sets. Each rotation can be tested for surjectivity using simple bit shifts that can happen in registers. In practice, this significantly

<sup>1</sup> The function would have to map exactly two keys to each of the  $n/2$  positions, which happens with probability  $\binom{n}{2 \dots 2} \left(\frac{n}{2}\right)^{-n} = e^{-n} 2^{n/2} \text{poly}(n)$ .

improves the construction time because fewer hash functions need to be evaluated. However, rotation fitting only affects a polynomial factor in the construction time, while other measures can contribute exponential factors.

### 4.1.2 Quad Split

The quad split technique reduces the amount of time spent on finding surjective seed candidates even more. It basically applies the idea of bipartite ShockHash on another level of the same data structure. Like in rotation fitting, we split the input set into two sets  $A$  and  $B$  using a constant 1-bit hash function. Now we can hash each of the two sets using *independent* hash functions. In particular, we can test all combinations of assigning some hash function to each of the two sets. This reduces the number of hash function evaluations significantly. To encode the combined seed, we use a pairing function again. In contrast to the bipartite tries in ShockHash, the hash functions cannot be exchanged, so we cannot assume that one seed is larger than the other. We therefore need a more general pairing function. The most fitting pairing function here is Szudzik's pairing function (see Section 2), which enumerates, for all  $k \in \mathbb{N}$ , all pairs in  $[k] \times [k]$  before moving on to pairs involving numbers bigger than  $k$ . This means that we can test all combinations of previous hash functions before having to evaluate the next one. In our implementation, we make sure to try hash function combinations in linear order in the value of the pairing function.

**Bit-parallel Compatibility Check.** We use bit patterns to determine the compatibility of two seeds efficiently. For both subsets, we linearly evaluate hash function seeds on both sets. For each seed, we determine a bit pattern for each subset that indicates positions that are hit by hash values. Let  $a$  and  $b$  be the bit patterns of the two subsets  $A$  and  $B$ . Then we can calculate the logical OR of  $a$  and all previous patterns from  $B$ . We perform the same check between  $b$  and all previous patterns from  $A$ , and also check  $(a \text{ OR } b)$ . If the result has all bits set to 1, we have found a surjective seed candidate. If it doesn't, we append each pattern to its corresponding list and determine the next pattern using the next hash function seed. In practice, using a sentinel element for this inner loop comparing bit patterns improves the performance significantly.

**Filtering.** The quad split technique described above significantly improves the running time because it drastically reduces the number of hash function evaluations. However, it does not improve the asymptotic running time much because all combinations still need to be tested. What makes this method interesting from a theoretical point of view is that we can again perform filtering here. The filter is less effective than on the main layer of the data structure, but it can still improve the asymptotic running time of the entire construction. Recall that we want to find surjective hash functions from  $n$  keys to  $\lceil n/2 \rceil$  positions here. We observe that if we fix one of the hash functions and this hash function does not hit one of the output values, the other hash function *has* to hit that output value. Therefore, we can skip testing the combination with all other hash functions that do not hit that output value. We can organize all candidate hash functions in a binary trie data structure based on the positions that are hit. Testing a new candidate hash function now boils down to traversing the tree. In theory, this is more efficient than testing all combinations. In practice, at least for the input sizes that we use in our experiments, the technique is likely slower because of constant factors, so we do not implement it. Preliminary experiments show that a version that uses bucketing based on the output values that are hit is not practical. While it is slightly faster



for extreme configurations with large  $n$ , it is usually slower than simply checking all bit patterns.

## 4.2 Variants and Enhancements

Until now, we have assumed that the input size  $n$  is an even number, but we now show how bipartite ShockHash can be enhanced to support uneven numbers of input keys as well. Additionally, we give further implementation tricks to improve the performance even more.

**Supporting Uneven  $n$ .** To support uneven numbers  $n$  of input keys, we can relax the bipartite property. The idea is that the output value  $\lceil n/2 \rceil$  can be hit by both hash functions, but each with half the probability. When then combining the two halves, it gets the normal probability like all other output values. For filtering candidate hash functions for surjectivity, the corresponding bit needs to be ignored – a seed candidate can be valid both if the bit is set or not set. While this is possible in theory, it is less elegant at query time. The problem is that both hash functions apply the reduced weight to the value  $\lceil n/2 \rceil$ , but one function gets shifted. Therefore, we need to mirror the shifted hash function so that the middle value receives the reduced weight. In our implementation we therefore simply accept that the output value  $\lceil n/2 \rceil$  is hit with twice the probability. Supporting uneven  $n$  then just boils down to rounding compile time constants to the right direction. When integrated into RecSplit, we expect uneven  $n$  to happen only once in every two buckets, so it has a negligible overhead of about  $1/2b$  bits per key.

**Isolated Keys Filter.** Bipartite ShockHash generates a set of surjective hash function candidates and then tests all combinations for orientability. By using an additional filter, we can speed up this test for orientability. The idea is to look at the case that a key is the only one mapping to a position. We refer to this key as *isolated* for that hash function seed. More formally, a key  $x$  is isolated using a hash function candidate  $h$ , if  $h^{-1}(h(x)) = \{x\}$ . If a key is isolated in *both* of the candidate hash functions, then in graph terminology this corresponds to a connected component with two nodes and one edge. Since each connected component of the final graph must have the same number of nodes and edges, there is then no need to perform the full test for orientability. We can determine bit patterns for each seed, indicating which of the keys are isolated. Then seed combinations can be ruled out using simple bit-parallel operations checking if the bit patterns are orthogonal. Note that the bit patterns used in the quad split technique refer to output positions (and have size  $n/2$ ), while the patterns here refer to input keys (and therefore have size  $n$ ). While this check saves the more complex check for orientability, this does not change the asymptotic running time much because all combinations still need to be compared. In theory, we could use a trie structure similar to the one in the quad split technique again. Then only seeds that have compatible isolated nodes need to be tested.

**Seed Cache.** Evaluating the hash functions can be a bottle-neck depending on the input size. An obvious idea is to cache the hash function output values of the seed candidates. Because the input sets and therefore the hash values are very small, we can store each hash value in a single byte. This makes the amount of space needed for each seed candidate relatively small.

**Parallelization.** When integrating bipartite ShockHash into the RecSplit framework, similar to how it is done in plain ShockHash, parallelization is obvious. Each bucket in RecSplit



can be handled independently, and there are so many buckets that a simple distribution of buckets to threads is likely well load-balanced. The main missing part for ShockHash parallelization is that the space-efficient retrieval data structure we use (BuRR [6]) is not parallelized. However, in the most space-efficient configurations, the retrieval data structure does not have a significant influence on the construction time. Therefore, our implementation of bipartite ShockHash only parallelizes the hash function search within buckets, based on SIMDRecSplit [4].

Bipartite ShockHash can also be parallelized using SIMD instructions. Compared to plain ShockHash, the bipartite version introduces more complex data structures. We achieve good performance improvements on large  $n$  by SIMD-parallelizing the comparison of candidate bit patterns. However, the other more involved data structures are harder to parallelize using SIMD because of more complex control flows. Therefore, we use SIMD only for checking lists of bit patterns, which is the main bottle-neck for large  $n$ . It is possible to use a small trick here: We can halve the size of the bit patterns, which doubles the number of SIMD lanes we can use. If there is a match, we then need to check the full pattern before continuing. Note also that even if the full pattern is never compared, this does not hurt correctness because the patterns are only used for filtering. However, on the machines we have tried this, the idea does not improve the performance. SIMDRecSplit [4] does not support the large leaf sizes that we can now achieve with bipartite ShockHash. Therefore, our SIMD implementation parallelizes only the leaves using bipartite ShockHash, not the splittings.

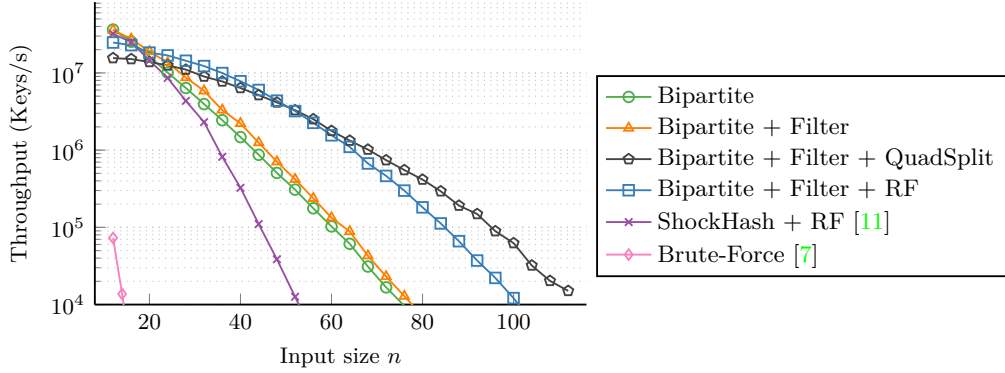
Finally, it is also possible to develop a hybrid CPU/GPU implementation. Many of the data structures used in bipartite ShockHash are more complex in terms of control flow and memory access. They are therefore hard to parallelize on the GPU. However, for large  $n$ , the majority of the construction time in bipartite ShockHash is spent on comparing bit patterns of hash function values with each other to check if they are compatible. Here we can use the massive parallelism of GPUs to produce hash function candidates by comparing many patterns in parallel. Then the CPU can perform the less frequent and more complex checks for orientability.

## 5 Experiments

In the following section, we describe experiments comparing bipartite ShockHash to the original implementation, as well as to competitors from the literature.

**Experimental Setup.** We run our experiments on an Intel i7 11700 processor with 8 cores and a base clock speed of 2.5 GHz. The machine runs Ubuntu 22.04 with Linux 5.15.0 and supports AVX-512 instructions. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. As input data, we use strings of uniform random length  $\in [10, 50]$  containing random characters except for the zero byte. Note that, as a first step, almost all compared codes generate a *master hash code* of each key using a high quality hash function. This makes the remaining computation independent of the input distribution. All experiments use a single thread.

**Construction.** Figure 4 shows different methods to generate seed candidates. Filtering based on isolated keys (see Section 4.2) makes the construction about two times faster. While rotation fitting already gives impressive speedups, our quad split technique (see Section 4.1.2) is even faster for large  $n$ . The quad split technique is about one order of magnitude faster than a basic implementation of bipartite ShockHash. For comparison, the plot also includes



■ **Figure 4** Construction throughput using different methods to come up with seed candidates. For comparison, the plot also includes Brute-Force search, as well as plain ShockHash.

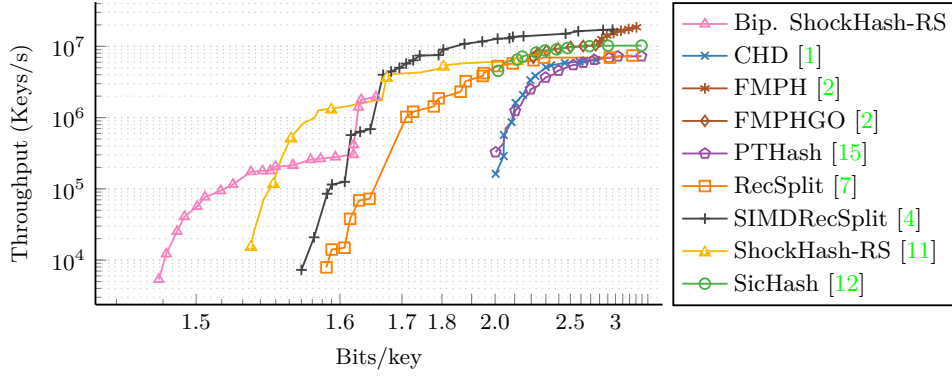
Brute-Force search, as well as the plain ShockHash implementation. It shows that, overall, bipartite ShockHash is up to about 3 orders of magnitude faster than the previous state of the art. Note that the comparison here is a bit unfair because different methods have different space overheads based on  $n$ . However, for larger  $n$ , the methods have almost the same space usage.

**Comparison with Competitors.** In the literature, there are many perfect hash function constructions. As with plain ShockHash, the input sets for bipartite ShockHash are still rather small ( $n \leq 128$ ). For a comparison with competitors, we therefore integrate it into the RecSplit [7] framework (see Section 2), just like plain ShockHash. RecSplit recursively divides the input set of size  $N$  into many smaller subsets, most of them having size  $n$ . We call this integrated variant *bipartite ShockHash-RS*. In the following, we always use the quad split technique with filtering based on isolated keys.

Figure 5 gives a Pareto front<sup>2</sup> of space usage vs construction performance for competitors from the literature. Note that we are mainly interested in the space usage configurations close to the theoretical lower bound. Therefore, the Figure uses a logarithmic x-axis. All of the competitors achieving below 1.8 bits per key use the RecSplit framework. This shows how seminal the RecSplit paper is, and how well the idea of the splitting tree works in practice. Bipartite ShockHash-RS becomes Pareto optimal for space usages less than 1.54 bits per key and is the first method to achieve a space usage of 1.489 bits per key in practice.

In addition to the Pareto front, we give a selection of configurations of the most space efficient competitors in Table 1. Given the same construction time, original RecSplit achieves a space usage of 1.584 bits per key, SIMDRecSplit a space usage of 1.560 bits per key and plain ShockHash-RS achieves a space usage of 1.524 bits per key. Bipartite ShockHash-RS reduces this to a space usage of only 1.489 bits per key. Therefore, making ShockHash bipartite reduces the distance to the lower bound from 0.081 bits per key to 0.046 bits per key, which is a reduction by 43%. Bipartite ShockHash-RS achieves a space usage within 3.3% of the lower bound for practically relevant input set sizes. Comparing configurations with the same space usage, plain ShockHash-RS generates a MPHf with 1.523 bits per key in about 19 minutes, while bipartite ShockHash-RS needs less than 1 minute to achieve the

<sup>2</sup> A point is on the Pareto front if there is no point from the same competitor that has simultaneously better space consumption and construction throughput.



■ **Figure 5** Pareto front of space usage vs construction performance. Note that both axes are logarithmic.  $N = 10$  million keys. We plot all Pareto optimal data points but only show markers for every second point to increase readability. Therefore, the lines might bend on positions without markers.

■ **Table 1** Query and construction performance of the most space efficient competitors.  $N = 5$  million keys. Space is given in bits per key and construction time is given in ns per key.

Method	Space	Constr.	Query
PTHash, $c=7.0$ , $\alpha=0.99$ , C-C	3.313	200 ns	20 ns
RecSplit, $n=14$ , $b=2000$	1.584	126 140 ns	97 ns
SIMDRecSplit, $n=14$ , $b=2000$	1.585	11 696 ns	109 ns
SIMDRecSplit, $n=16$ , $b=2000$	1.560	137 919 ns	99 ns
ShockHash-RS, $n=30$ , $b=2000$	1.582	797 ns	121 ns
ShockHash-RS, $n=39$ , $b=2000$	1.556	1 813 ns	123 ns
ShockHash-RS, $n=58$ , $b=2000$	1.523	112 072 ns	121 ns
Bip. ShockHash-RS, $n=64$ , $b=2000$	1.524	5 724 ns	131 ns
Bip. ShockHash-RS, $n=104$ , $b=2000$	1.496	24 406 ns	121 ns
Bip. ShockHash-RS, $n=128$ , $b=2000$	1.489	188 041 ns	113 ns

same space consumption. This makes bipartite ShockHash-RS about 20 times faster than the plain version. The reason why bipartite ShockHash-RS has a lower speedup than the base case is that for this configuration, about 80% of the construction time is spent in the splitting tree. The most relevant configurations of bipartite ShockHash-RS, where the base case dominates the construction time, are entirely impractical for plain ShockHash-RS.

Table 1 also gives the query time of different configurations of competitors. Note that while the splitting tree at the core of RecSplit leads to impressive space usage, it is rather slow to traverse. Therefore, methods inside the RecSplit framework, which are focused mainly on space consumption, have rather slow (but still constant-time) queries. This is compared to less space efficient competitors numerous times in the literature [4, 11, 15]. In the Table, we therefore only give PTHash as an example, which is currently the MPHF with the fastest queries. In general, we expect any ShockHash implementation to have slower queries than RecSplit because of the additional access to a retrieval data structure. Compared to plain ShockHash-RS, our bipartite implementation loses about 10% of the query performance at the same space usage. This is due to un-pairing seeds and having to do case distinctions based on the partitions and on the subset during quad split. However, bipartite ShockHash-RS

has the potential to make queries faster by increasing the size of the leaf nodes, as shown in the measurements. This reduces the time spent on traversing RecSplit’s splitting tree data structure, which is a major bottleneck for the queries.

## 6 Conclusion and Future Work

With bipartite ShockHash, we have presented an extension of ShockHash that significantly improves the space consumption that can be achieved in practice. The idea is to generate the ShockHash graph by two independent hash functions and testing all pairwise combinations of them. This significantly reduces the number of hash functions that need to be evaluated. On top, filtering the seed candidates provides exponential construction time speedups. In itself, the construction is up to 3 orders of magnitude faster than plain ShockHash. Integrating bipartite ShockHash into the RecSplit framework to support larger input sets, we get bipartite ShockHash-RS. Even though bipartite ShockHash-RS works best far beyond the smallest practical configurations plain ShockHash-RS, it is already 20 times faster than the smallest configuration of plain ShockHash-RS.

**Future Work.** This paper only describes the algorithmic ideas and gives initial experiments. An important next step therefore is a formal analysis of bipartite ShockHash that proves the properties we only used intuitively in this paper. Additionally, a hybrid parallelization of the technique using a GPU would be an interesting future direction. For full utilization of SIMD hardware, a further next step would be to work on SIMD-parallel splittings in the RecSplit framework that support the large leaf sizes that we can now achieve with bipartite ShockHash.

---

## References

- 1 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.
- 2 Piotr Beling. Fingerprinting-based minimal perfect hashing revisited. *ACM Journal of Experimental Algorithmics*, 2023.
- 3 Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *FOCS*, pages 182–193. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00026.
- 4 Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. High performance construction of RecSplit based minimal perfect hash functions. *CoRR*, abs/2212.09562, 2022. doi:10.48550/ARXIV.2212.09562.
- 5 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.
- 6 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.
- 7 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.
- 8 Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM, 2014. doi:10.1145/2674005.2674994.

- 9 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](https://doi.org/10.1145/133160.133209).
- 10 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:[10.1145/828.1884](https://doi.org/10.1145/828.1884).
- 11 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Shockhash: Towards optimal-space minimal perfect hashing beyond brute-force. *arXiv preprint arXiv:2308.09561*, 2023.
- 12 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Sichash - small irregular cuckoo tables for perfect hashing. In *ALENEX*, pages 176–189. SIAM, 2023. doi:[10.1137/1.9781611977561.CH15](https://doi.org/10.1137/1.9781611977561.CH15).
- 13 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](https://doi.org/10.4230/LIPICs.SEA.2017.25).
- 14 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](https://doi.org/10.1007/978-3-319-07959-2_12).
- 15 Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. doi:[10.1145/3404835.3462849](https://doi.org/10.1145/3404835.3462849).
- 16 Matthew Szudzik. An elegant pairing function. In *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*, pages 1–12, 2006.