Non-minimal k-perfect hashing: Tight lower bounds and an application to fast static hash tables

Ragnar Groot Koerkamp
Karlsruhe Institute of Technology, Germany

Stefan Hermann
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* (minimal PHF) is a data structure mapping a static set of n keys to n bins without collisions. Two natural generalizations are minimal k-PHFs where n keys are mapped to n/k bins of capacity k each, and (non-minimal) PHFs with load factor $\alpha < 1$ where the number of bins is increased by a factor of $1/\alpha$, resulting in spare capacity.

While there has been a recent surge of interest in perfect hashing generally, non-minimal k-PHFs have not been systematically studied despite a natural use case of speeding up static hash tables: The idea is that a small cache-resident k-PHF maps each key x to a cache-line-sized bin of capacity k where x resides. Ideally, this yields a branchless lookup operation with a single cache miss working at high load factors for positive and negative queries alike.

Our main theoretical contribution is to determine tight space lower bounds for k-PHFs for all pairs of $\alpha \in (0, 1]$ and $k \geq 1$. It turns out that combining $\alpha < 1$ and $k \geq 2$ drastically reduces the space of k-PHFs, e.g. for $(k, \alpha) = (16, 0.8)$ the space lower bound is 0.027 bits per key while for $(k, \alpha) = (16, 1.0)$ and $(k, \alpha) = (1, 0.8)$ the lower bounds are higher by factors of ≈ 8 and ≈ 32, respectively. On the practical side, we develop a k-PHF based on PtrHash and tune it for use in static hash tables. Empirically, our implementation produces k-PHFs of size roughly 50% above the lower bound. A static hash set based on this k-PHF is consistently at least as fast as other hash sets for negative and mixed queries. On two of the three tested architectures it achieves up to 1.5× speedup for large $n \geq 30M$ where a 1-PHF does not fit in cache.

2012 ACM Subject Classification Theory of computation → Bloom filters and hashing; Theory of computation → Data structures design and analysis; Information Systems → Point lookups

Keywords and phrases Compressed Data Structures, k-Perfect Hashing, Hash Table, Space Lower Bound

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.18

Related Version *Extended version*: `https://arxiv.org/abs/2607.07257` [26]

Supplementary Material *Software*: `https://github.com/RagnarGrootKoerkamp/static-hash-sets`

1 Introduction

Given a universe of u keys and an input set of n keys, a k-perfect hash function (PHF) maps the input keys to b *bins* such that there are at most k keys in each bin. The challenge is to represent the k-PHF as a data structure without storing the input set itself. We refer to $\alpha := \frac{n}{bk}$ as the *load* of a k-PHF and call a k-PHF *minimal* when $\alpha = 1$. Table 1, as well as the survey [35], give an overview of construction techniques and lower bounds. In this paper, we initiate the study of the most difficult case of non-minimal k-PHF with $k \geq 2$.

$k = 1$, $\alpha = 1$	Bucket Placement [19, 2, 47, 28, 25, 4], Recursive Splitting [16, 38, 5], Fingerprinting [3, 10, 40, 34], Hypergraph / Retrieval [7, 14, 43, 6, 20, 52, 36, 37, 27] **Lower Bound:** $n \cdot \log_2(e)$ [44]
$k = 1$, $\alpha < 1$	Bucket Placement [19, 2, 47, 28, 25, 4], Hypergraph / Retrieval [11, 36] **Lower Bound:** $n \cdot \left(\log_2(e) + \left(\frac{1}{\alpha} - 1\right)\log_2(1-\alpha)\right)$ [2]
$k \geq 2$, $\alpha = 1$	Bucket Placement & Recursive Splitting & Thresholding [29] **Lower Bound:** $n \cdot \left(\log_2(e) - \log_2(k^k/k!)/k\right)$ [42, 2, 31]
$k \geq 2$, $\alpha < 1$	Bucket Placement [2] and [Section 3] **Lower Bound (new):** $n \cdot \left(\log_2\left(\frac{\lambda e}{\alpha k}\right) + \frac{\log_2 \zeta}{\alpha k}\right)$ [Theorem 8] where λ s.t. for $\mathbb{E}_{Y \sim \mathrm{Pois}(\lambda)}[Y \mid Y \leq k] = \alpha k$ and $\zeta := e^{-\lambda}/\Pr[Y \leq k]$.

Table 1 Overview of static techniques and lower bounds. For the lower bounds we assume a large universe and omit lower order terms. For details about the techniques, we refer to the survey [35].

We derive tight lower bounds on the space of a non-minimal k-PHF for all pairs of α and k (Section 2), extend PtrHash [25] into k-PtrHash, a fast and practical k-PHF with about 50% space overhead (Section 3), and apply this k-PHF to obtain a fast static hash set for large inputs (Section 4).

Theory: Tight lower bounds. One way to construct a PHF is via a brute force algorithm: try out seeds $s = 0, 1, \ldots$ of a hash function that randomly maps the keys to the bins, until one is found that maps at most k keys to each bin. Once found, the data structure stores s. Storing s takes roughly $\log_2 \frac{1}{q}$ bits, where q is the success probability of a single seed. For the cases where $k = 1$ *or* $\alpha = 1$, it was known that brute force is space optimal up to lower order terms [2]. The first theoretical contribution of this paper is to extend this result to all combinations of α and k:

▶ **Theorem 1** (informal version of Theorem 7). *Assuming the universe of keys is sufficiently large, the brute force algorithm is space optimal for all* $0 < \alpha \leq 1$ *and* $k \geq 1$.

Hence, tight lower bounds can be attained by determining the probability q that a random function $f : S \to [b]$ from a set S of n keys to b bins is k-perfect. Equivalently q is the ratio of the number P of k-perfect functions and the number b^{-n} of all functions.

For $\alpha \in (0, 1]$ and $k = 1$ we have $P = \frac{b!}{(b-n)!}$ by standard counting arguments. A simple calculation using Stirling's approximation then yields the first two lower bounds shown in Table 1. The case of $k \geq 2$ and $\alpha = 1$ is also easy: We have $P = \binom{n}{k\ k\ \cdots\ k} = \frac{n!}{(k!)^b}$ from which the third bound in Table 1 follows.

The case of $k \geq 2$ and $\alpha < 1$ is more difficult. The issue is that non-minimal k-perfect functions are more varied with an unknown number of bins of the various loads in $\{0, \ldots, k\}$. While we can still write a formula like

$$P = \sum_{\substack{x_1,\ldots,x_b \in \{0,\ldots,k\} \\ x_1+\cdots+x_b=n}} \binom{n}{x_1\ x_2\ \ldots\ x_b},$$

such a formula is not useful, for instance it does not reveal the asymptotic behaviour of q. Previous work has also stated that the case of $k \geq 2$ and $\alpha \in (0, 1)$ appears non-trivial [2]. Using a new approach, our main theoretical contribution is to determine the success probability q for any combination of k and α. This gives:

▶ **Theorem 2** (informal version of Theorem 8). *For any $\alpha \in (0,1)$ and $k \geq 1$, the tight space lower bound for non-minimal k-PHF is*

$$n \cdot \left(\log_2 \left(\frac{\lambda e}{\alpha k} \right) + \frac{\log_2 \zeta}{\alpha k} \right)$$

where $\lambda = \lambda(\alpha, k)$ and $\zeta = \zeta(\alpha, k)$ are as defined in Table 1.

The theorem implies that the optimal brute-force algorithm results in a situation where the number of keys in each bin follows a *truncated Poisson distribution* with parameter λ (taking only values $\leq k$) and expectation αk. (The distribution is slightly skewed by the global condition that the bin loads sum to n.)

We also derive a second result on the space requirement of k-PHFs that neglects constant factors but has the advantage of yielding closed form expressions. The result involves the *slack* $s = \lfloor (1-\alpha)k + 1 \rfloor$, which is intuitively the average number of free slots per bin. The result reveals two regimes, depending on whether the slack is higher or lower than the standard deviation $\sqrt{k}$ of bin loads in random assignments.

▶ **Theorem 3** (Informal version of Theorem 10). *Let $S(\alpha, k)$ be the tight space lower bound for a k-PHF in bits per bin. There are the following cases:*

If $s \ll \sqrt{k}$ then $S(\alpha, k) = \Theta(\log(\sqrt{k}/s))$.
If $s = \Theta(\sqrt{k})$ then $S(\alpha, k) = \Theta(1)$.
If $s \gg \sqrt{k}$ then $S(\alpha, k) = \exp(-\Theta(s^2/k))$.

Practice: k-PHF using k-PtrHash. The three relevant performance metrics of a PHF are construction throughput, space usage, and query throughput. A variety of algorithms exists for the case $k = 1$, covering a large spectrum of trade-offs between the three metrics. While some techniques are extremely close to the lower bound [38] ($\alpha = 1$), others offer fast construction and queries [25, 4]. We refer to the recent survey on minimal 1-PHFs for details [35]. A number of methods was recently generalized to minimal k-PHFs for $k \geq 2$ [29], while the case ($k \geq 2, \alpha < 1$) has only received minimal attention [2]. The only technique that has so far been applied to all combinations of α and k is *bucket placement*. In this paper, we implement k-PtrHash (Section 3), a natural generalization of the PtrHash bucket placement technique [25] that is tuned for high query throughput and space efficiency.

Bucket placement works by first partitioning the key set into small buckets. Buckets are then processed one-by-one: For each bucket, a *seed* (of a hash function) is found where all keys of that bucket are hashed (placed) to the bins such that there are at most k keys in each bin. For $k = 1$, this allows a natural greedy algorithm that chooses for each bucket the first working seed, and this method was also used for $k \geq 2$ in [29] and [2]. For $k \geq 2$, however, this is suboptimal and makes placing future keys harder than needed: whereas for $k = 1$ the only "state" is the number of full bins, for $k \geq 2$, there is a "quality" to the distribution of the keys so far. For example, consider $k = 2$ and $\alpha = 1$. At the halfway point, we might have filled exactly one of the two slots in each bin, in which case there is still a lot of freedom to map the remaining keys. If, on the other hand, exactly half of the bins are filled while the other half is empty, mapping the remaining keys is much more restricted. Thus, one should prefer seeds that result in a more even "spread" of the keys over the bins, and we introduce a *scoring function* to express this preference.

Our implementation k-PtrHash reaches as low as 50% space overhead over the lower bound, while reaching a throughput of over 300 million queries per second.

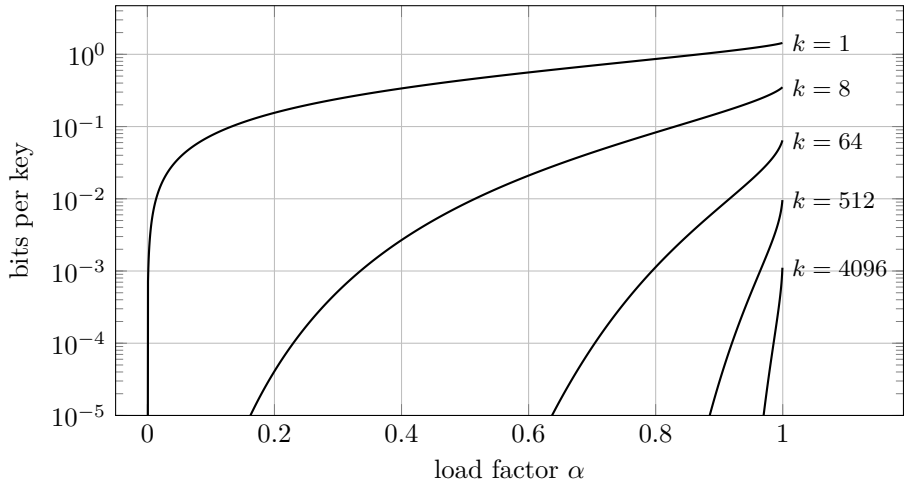

Figure 1 Space lower bound for different load factors and k.

Table 2 Space lower bounds for different k and α in bits per key, as computed by the script in Appendix B.

α	$k=1$	$k=2$	$k=4$	$k=8$	$k=16$	$k=32$	$k=64$
0.20	0.154983	0.031120	0.002591	0.000041	≈ 0	≈ 0	≈ 0
0.40	0.337247	0.112213	0.024445	0.002680	0.000082	≈ 0	≈ 0
0.60	0.561410	0.244721	0.084103	0.020956	0.003182	0.000199	0.000002
0.80	0.862213	0.456247	0.210448	0.082945	0.026996	0.006774	0.001132
0.90	1.073592	0.621795	0.326312	0.154184	0.064989	0.024040	0.007553
0.95	1.215225	0.739728	0.416518	0.216630	0.103742	0.045522	0.018156
0.99	1.375585	0.880564	0.533189	0.306808	0.168119	0.087660	0.043398
1.00	1.442695	0.942695	0.588936	0.355096	0.208329	0.119672	0.067619

Application: a static hash table based on k-PHF. A natural application of k-PHFs are static hash sets[1] (and tables): Given a set of keys, one can first construct a k-PHF and then place the keys into their bins. A natural choice for k is the number of keys that fit into a cache line. Given a query key, we evaluate the k-PHF, retrieve the single corresponding cache line and then answer the query using SIMD. A non-minimal 8-PHF with $\alpha = 0.9$ requires $7\times$ times less space than a 1-PHF with $\alpha = 0.9$. The reduced space usage allows the k-PHF to fit in the cache for larger inputs. For large inputs, the k-PHF-set is consistently the fastest for negative and mixed queries and achieves up to $1.5\times$ higher query throughput on two of the three tested architectures (Section 4).

2 Tight Lower Bounds for non-minimal k-PHFs

In this section we determine tight lower bounds on the space requirement of non-minimal k-PHFs. In Section 2.1 we show that a simple brute force algorithm is space-optimal, which

[1] In fact, the practical part of this paper was motivated by Deacon [13], which is a tool that stores a static subset of 300 million of the k-mers in a human genome and then queries this set to determine whether pieces of DNA are human or not.

is neither surprising nor particularly difficult. What *is* difficult is computing the expected space usage of that brute force algorithm, which we do in Section 2.2. Our resulting formula is exact (up to lower order terms) but unwieldy. We therefore also derive explicit formulas that neglect constant factors in Section 2.3. These use completely different arguments.

2.1 Brute Force is Space-Optimal

Once constructed, a k-PHF behaves like a function $f : [u] \to [b]$ where $f(x)$ is the bin returned for the key x. The k-PHF must be *compatible* with its input set $S \subseteq [u]$, meaning $|f^{-1}(i) \cap S| \leq k$ for all $i \in [b]$. Let $\text{comp}(f)$ be the number of input sets of size n that are compatible with f. Assume for simplicity that b divides u and call f *balanced* if $|f^{-1}(1)| = \ldots = |f^{-1}(b)|$.

Consider the following intuitive statement.

▶ **Lemma 4.** $\text{comp}(f)$ *is maximised if* f *is balanced.*

By standard arguments it suffices to prove the following simpler lemma.

▶ **Lemma 5.** *Consider a set of* u *distinguishable balls of various colours, including* R *red balls and* B *blue balls with* $R \geq B + 2$*. Consider the number of subsets of size* n *that involve at most* k *balls of each colour. This number is non-decreasing if we recolour a red ball blue.*

Proof. It suffices if we prove the lemma assuming that red and blue are the only available colours. This is because for whatever subset S of balls of the other colours we consider, the number of ways to extend S to a valid subset overall is non-decreasing if we do the recolouring.

Hence assume $u = R + B$ and fix a special red ball that is to be recoloured. We may also assume $B \geq k$ and $k < n \leq 2k$ as otherwise the claim is trivial. The subsets of n balls fall into the following categories.

- neutral subsets where the special ball is not drawn,
- neutral subsets where the colour of the special ball does not make a difference,
- $\binom{R-1}{k}\binom{B}{n-k-1}$ good subsets where the special ball would have been a red ball to many,
- $\binom{R-1}{n-k-1}\binom{B}{k}$ bad subsets where the special ball will be a blue too many.

We compute that the good cases outnumber the bad cases.

$$\frac{\text{good}}{\text{bad}} = \frac{\binom{R-1}{k}\binom{B}{n-k-1}}{\binom{R-1}{n-k-1}\binom{B}{k}} = \frac{(R-1-n+k+1)!(B-k)!}{(R-1-k)!(B-n+k+1)!}$$
$$= \frac{(B-k)\cdot\ldots\cdot(B-n+k+2)}{(R-1-k)\cdot\ldots\cdot(R-1-n+k+2)} \geq 1\cdot\ldots\cdot 1 = 1$$ ◀

Let now $q_{\text{bal}} = q_{\text{bal}}(u, n, b, k)$ be the probability that a uniformly random input set is compatible with a fixed balanced function, or equivalently[2], the probability that a fixed input set is compatible with a uniformly random balanced function.

▶ **Theorem 6.** *The* balanced brute force algorithm *(see below) constructs a* k*-PHF with expected space* $\log_2 \frac{1}{q_{\text{bal}}} + \mathcal{O}(1)$*, with* q *as defined above, and no (randomised) construction can undercut this bound on expected space by more than* $\mathcal{O}(1)$ *bits.*

[2] To see the equivalence, first define q as the probability that a random input set is compatible with a random balanced function. Then note the symmetry between all input sets and the symmetry between all balanced functions. These symmetries imply that we can condition on any input set or any balanced function without affecting q.

Proof. The balanced brute force algorithm tries a sequence $f_1, f_2, \ldots$ of uniformly random balanced functions and stores the smallest index S for which f_S is compatible with the input set. This seed has distribution $S \sim \mathrm{Geom}(q_{\mathrm{bal}})$ and can be stored in expected space $\log_2 \frac{1}{q_{\mathrm{bal}}} + \mathcal{O}(1)$ using Rice encoding.

Now we show that no other construction can do significantly better. Consider any deterministic construction $\mathcal{D}$ first. Let $\mathcal{F} \subseteq [b]^{[u]}$ be the family of functions that $\mathcal{D}$ may produce over all possible input sets combined. Any $f \in \mathcal{F}$ satisfies $\mathrm{comp}(f) \leq q_{\mathrm{bal}} \cdot \binom{u}{n}$ by Lemma 4 and the definition of q_{bal}. Moreover, $\sum_{f \in \mathcal{F}} \mathrm{comp}(f) \geq \binom{u}{n}$ as every input set must be compatible with some $f \in \mathcal{F}$. This implies $|\mathcal{F}| \geq \frac{1}{q_{\mathrm{bal}}}$. To represent every possible $f \in \mathcal{F}$, at least $\log_2 |\mathcal{F}| \geq \log_2 \frac{1}{q_{\mathrm{bal}}}$ bits are needed in the worst case.

It is not hard to see that if the input is uniformly random (rather than worst case), the expected space requirement drops by at most $\mathcal{O}(1)$ bits.[3] By Yao's Theorem, $\log_2 \frac{1}{q_{\mathrm{bal}}} - \mathcal{O}(1)$ is therefore also a lower bound for the expected space requirement of any randomised algorithm. ◀

Now consider the related *simple brute force* algorithm that tries uniformly random functions that are not necessarily balanced. Under a weak assumption simple brute force is also optimal.

▶ **Theorem 7.** *Assume $u \geq n^2/\alpha$. Then simple brute force is space optimal in the same sense as balanced brute force (see Theorem 6).*

The proof is standard and found in the appendix Appendix A. To see that a lower bound on u is required, consider the silly edge case of $u = n$ and $\alpha = k = 1$. The identity function is minimal perfect and requires no space to store, while simple brute force requires $\log_2 \frac{1}{n!} \approx 1.44n$ bits.

2.2 Precise Formulas and the Success Probability of Simple Brute Force

In this section we approximate the probability q_{simp} that a uniformly random function is k-perfect on a given input set. This leads to the following result.

▶ **Theorem 8.** *Let $k, n, u \in \mathbb{N}$ and $\alpha \in (0, 1)$ with $u \geq n^2/\alpha$. Let λ be such that $\mathbb{E}_{Y \sim \mathrm{Pois}(\lambda)}[Y | Y \leq k] = \alpha k$ and $\zeta := e^{-\lambda} / \Pr[Y \leq k]$. Then the best possible space requirement for k-PHFs with load factor α is*

$$\underbrace{\log_2 \frac{\lambda e}{\alpha k} + \frac{\log_2 \zeta}{\alpha k}}_{S(\alpha,k):=} \pm \widetilde{\mathcal{O}}(\tfrac{1}{n}) \text{ bits per key where } \widetilde{\mathcal{O}} \text{ ignores terms only depending on } \alpha \text{ and } k.$$

Note that $S(\alpha, k)$ is independent of n and u. What makes the formula a bit unwieldy is that ζ and λ are non-explicit functions of α and k. Numerical methods (see Appendix B) nevertheless allow us to derive the values in Figure 1 and Table 2.

The main ingredient for proving Theorem 8 is the following lemma.

▶ **Lemma 9.** *In the setting of Theorem 8 we have* $q_{\mathrm{simp}} = \left(\frac{\alpha k}{e\lambda}\right)^n \zeta^{-b} \cdot \widetilde{\Theta}(1)$.

[3] This uses that if $Y \sim \mathrm{Unif}(\{1, \ldots, N\})$ then $\mathbb{E}[\log_2 Y] = \log_2 N - \mathcal{O}(1)$.

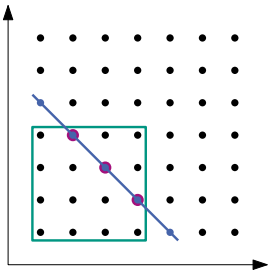

Figure 2 The two dimensional illustration uses $b = 2$, $k = 4$ and $n = 5$. The multinomial $(X_1, \ldots, X_b)$ is distributed on the blue "diagonal", since $X_1 + \ldots + X_b = n$ is fixed. We wish to compute the probability that $(X_1, \ldots, X_b)$ falls inside the green box (is "k-perfect"). To help with this, we consider a sequence $(Z_1, \ldots, Z_b)$ of truncated Poisson random variables that automatically fall within the box, but not on the diagonal.

Proof of Theorem 8. By Theorem 6, the best possible space requirement for k-PHFs is $\log_2 \frac{1}{q_{\text{bal}}} \pm \mathcal{O}(1)$ and by Theorem 7 we have $q_{\text{simp}} = \Theta(q_{\text{bal}})$ when $u \geq n^2/\alpha$ as we assume. We obtain the desired result from the claim of Lemma 9 by taking $\log_2 \frac{1}{\cdot}$ on both sides, using $b = \frac{n}{\alpha k}$, and dividing by n. ◀

Proof of Lemma 9. Consider a uniformly random function $f : S \to [b]$ where S is an input set of size n. For $i \in [b]$, let $X_i := f^{-1}(i)$ be the number of keys assigned to bin i. Clearly $\vec{X} := (X_1, \ldots, X_b)$ follows a multinomial distribution (the bin loads after "throwing n balls into b bins"). Let $R := \{(x_1, \ldots, x_b) \in \{0, \ldots, k\} \mid x_1 + \ldots + x_b = n\}$ be the set of possible outcomes for $\vec{X}$ that correspond to f being k-perfect on S. In particular we have $q_{\text{simp}} = \Pr[\vec{X} \in R]$. For any particular $\vec{x} = (x_1, \ldots, x_n) \in R$ we have

$$\Pr[\vec{X} = \vec{x}] = \binom{n}{x_1 \ \ldots \ x_b} b^{-n} = \frac{n!}{x_1! \cdot \ldots \cdot x_b!} b^{-n}$$

and by summing over all $x \in R$ we obtain

$$q_{\text{simp}} = \Pr[\vec{X} \in R] = \sum_{\vec{x} \in R} \Pr[\vec{X} = \vec{x}] = \frac{n!}{b^n} \sum_{\vec{x} \in R} \prod_{i=1}^{b} \frac{1}{x_i!}. \tag{1}$$

We will now consider a second random sequence $\vec{Z} = (Z_1, \ldots, Z_b)$ such that $\vec{Z}$ conditioned on R has the same distribution as $\vec{X}$ conditioned on R. See Figure 2 for an illustration. Concretely let $Z_1, \ldots, Z_b$ be independent with probability mass function

$$\Pr[Z = i] = \begin{cases} 0 & i > k \\ \zeta \frac{\lambda^i}{i!} & i \leq k. \end{cases} \tag{2}$$

where λ is chosen such that $\mathbb{E}[Z_i] = \alpha k$ and ζ is a normalisation factor. Let $\vec{Z} = (Z_1, \ldots, Z_b)$ and $N_Z = \sum_{i=1}^{b} Z_i$. By construction, the events $\vec{Z} \in R$ and $N_Z = n$ are equivalent. For any $\vec{x} \in R$ we can compute

$$\begin{aligned} \Pr[\vec{Z} = \vec{x} \mid N_Z = n] &= \frac{\Pr[\vec{Z} = \vec{x} \wedge N_Z = n]}{\Pr[N_Z = n]} = \frac{\Pr[\vec{Z} = \vec{x}]}{\Pr[N_Z = n]} = \frac{\prod_{i=1}^{b} \Pr[Z_i = x_i]}{\Pr[N_Z = n]} \\ &= \frac{\prod_{i=1}^{b} \zeta \frac{\lambda^{x_i}}{x_i!}}{\Pr[N_Z = n]} = \frac{\zeta^b \lambda^n}{\Pr[N_Z = n]} \prod_{i=1}^{b} \frac{1}{x_i!}. \end{aligned}$$

By summing this equation over all $\vec{x} \in R$ we get

$$1 = \frac{\zeta^b \lambda^n}{\Pr[N_Z = n]} \sum_{\vec{x} \in R} \prod_{i=1}^{b} \frac{1}{x_i!} \tag{3}$$

As $\mathbb{E}[N_Z] = b \cdot \mathbb{E}[Z_1] = b\alpha k = n$ the probability of the likely event $N_Z = n$ seems insignificant compared to the exponential terms. Indeed, using a local limit theorem for discrete random variables [50, Theorem 1.4] (originally in [22]) gives $\Pr[N_Z = n] = \Theta(1/(\sigma\sqrt{b}))$ where $\sigma^2 = \mathrm{Var}(Z_1)$. We may hence use $\Pr[N_Z = n] = \widetilde{\Theta}(1/\sqrt{n})$. We now rearrange Equation (3) for $\sum_{\vec{x}\in R} \prod_{i=1}^{b} \frac{1}{x_i!}$ and plug the result into Equation (1). Using Stirling's approximation we have

$$q_{\text{simp}} = \frac{n!}{b^n} \cdot \frac{\Pr[N_Z = n]}{\zeta^b \lambda^n} = \frac{n!}{b^n \lambda^n} \zeta^{-b} \cdot \widetilde{\Theta}(1/\sqrt{n})$$
$$= \left(\frac{n}{eb\lambda}\right)^n \cdot \zeta^{-b} \cdot \widetilde{\Theta}(1) = \left(\frac{\alpha k}{e\lambda}\right)^n \cdot \zeta^{-b} \cdot \widetilde{\Theta}(1). \qquad \blacktriangleleft$$

2.3 Tight Space Lower Bounds up to Constant Factors

We now derive simple *explicit* formulas for the space requirement of k-PHFs that neglect constant factors. In the following we use $s := \lfloor (1-\alpha)k + 1 \rfloor = |\mathbb{N} \cap [\alpha k, \dots, k]|$ to refer to the *slack*, which is similar to the number $(1-\alpha)k$ of spare slots per bin, though the definition ensures $s \geq 1$ even for $\alpha = 1$.

▶ **Theorem 10.** *Let $k \in \mathbb{N}$, $\alpha \in [\frac{1}{2}, 1]$, $s := \lfloor (1-\alpha)k + 1 \rfloor$ and $n \geq n_0(k, \alpha)$ sufficiently large. Assume $u \geq n^2/\alpha$. The best possible space requirement for k-PHFs with load factor α is*

for $s = \Theta(\sqrt{k})$:	$\Theta(1)$ *bits per bin*	*(total of $\Theta(b) = \Theta(\frac{n}{\alpha k})$ bits),*
for $s \ll \sqrt{k}$:	$\Theta(\log(\sqrt{k}/s))$ *bits per bin*	*(total of $\Theta(b \log(\sqrt{k}/s))$ bits),*
for $s \gg \sqrt{k}$:	$\exp(-\Theta(s^2/k))$ *bits per bin*	*(total of $\Theta(b \exp(-\Theta(s^2/k)))$ bits).*

Recall that the space requirement of simple brute force is optimal at $\log_2 \frac{1}{q_{\text{simp}}} \pm \Theta(1)$ bits by Theorem 7 where q_{simp} is the probability that a fully random assignment of n keys to b bins results in loads $(X_1, \dots, X_b)$ satisfying $\max_{i\in[b]} X_i \leq k$. Note that, while they are not independent, individually the loads X_i have distribution $\mathrm{Bin}(n, \frac{1}{b})$.

The intuition for the bounds in Theorem 10 is as follows. In the *heavily loaded case* ($s \ll \sqrt{k}$) the extra number s of slots per bin is much less than the standard deviation $\sqrt{k}$ of each X_i. The probability that a bin is neither overflowing nor significantly underloaded is $\Pr[X_i \in [k-2s, k]] = \Theta(s/\sqrt{k})$. This suggests $q_{\text{simp}} \approx \Theta((s/\sqrt{k})^b)$ leading to a space of $\log_2 \frac{1}{q_{\text{simp}}} = \Theta(b \log(\sqrt{k}/s))$ as claimed.

In the *lightly loaded case* ($s \gg \sqrt{k}$) the probability that an individual bin overflows is $\exp(-\Theta(s^2/k))$ by Chernoff bounds and minimal intervention at the overflowing bins requires $\Theta(1)$ bits to fix each of them. The lower bound will exploit the following lemma, which happens to not be wasteful in this case.

▶ **Lemma 11.** $\Pr[\max X_i \leq k] \leq \Pr[X_1 \leq k]^b$.

Proof. We exploit that $X_1, \dots, X_b$ are what is known as *negatively associated* [30, Section 3.1]. This implies that they are *negative lower orthant dependent* [30, Section 2, Property 3], which is the name of the property we have stated. ◀

Due to space constraints the lengthy proof of Theorem 10 was moved to Appendix C.

3 Extending PtrHash to k-perfect hashing

We now extend PtrHash [25] to non-minimal k-perfect hashing.

3.1 The original PtrHash

We briefly recall how PtrHash works. Like its precursors PTHash [47] (which improved FCH [19] and CHD [2]) and PHOBIC [28], PtrHash is based on *bucket placement*. We are given n keys $\{k_1, \ldots, k_n\}$, and let $\alpha \leq 1$ be the target load factor, so that the 1-PHF injectively maps the n keys to $b = n/\alpha$ slots. In PtrHash, values between n and n/α are then *remapped* into $[n]$, but this is not needed for *non-minimal* PHFs. The keys are hashed using a 64-bit hash h_1 and non-uniformly mapped to $B = \lceil n/\lambda \rceil$ *buckets* of average size λ (in practice around 3) using a *bucket assignment function* $\gamma : [0,1) \mapsto [0,1)$ such as $\gamma(x) = x^2$. The data structure allocates an 8-bit *seed* s_j for each bucket j that controls the slot that the keys finally hashes to using a second hash function h_2. To summarize in formulas:

$$\begin{aligned} \text{bucket}(k_i) &:= \lfloor B \cdot \gamma(h_1(k_i)/2^{64}) \rfloor, \\ \text{seed}(k_i) &:= s_{\text{bucket}(k_i)}, \\ \text{slot}(k_i) &:= h_2(k_i, \text{seed}(k_i)) \bmod b. \end{aligned} \tag{4}$$

Construction. The construction algorithm first distributes the keys over the buckets, and then processes the buckets from large to small. For each bucket, it tries seeds from 0 to $2^8 - 1$ until one is found such that the corresponding slots are all still free. The first working seed is greedily chosen and the corresponding slots are marked as taken.

Hash-evict. If none of the seeds work, the seed with the minimal number of colliding slots is found and fixed. Colliding slots are handled by *evicting* the keys that previously mapped there: the seed for the colliding buckets is unset and these buckets are pushed on a queue. For each bucket in the queue, a new search for a seed is done, until the queue becomes empty. Conceptually, this eviction strategy is similar to cuckoo-hashing [46] as it spreads the complexity of hard buckets over multiple seeds. This also ensures that the seed bits for easy-to-place buckets are also used efficiently. Furthermore, queries remain as simple as a lookup of a single seed.

3.2 Greedy k-PHF

We naturally extend PtrHash to the case of k-perfect hashing by replacing $b = \lceil n/\alpha \rceil$ *slots* by $b = \lceil n/\alpha/k \rceil$ *bins* consisting of k slots each, similar to [2]. Whereas before it was sufficient to track which slots are already taken, now we count for each bin how many of its slots are taken. For each bucket, we greedily choose the first seed for which none of the mapped-to bins overflows, similar to the bucket placement method described in [29].

Choosing parameters. Given α and k, we compute the space lower bound ℓ in bits per key using Theorem 8. Then, we choose the target space usage as e.g. 2ℓ, so that each bucket of size 8 bits should contain on average $\lambda = 8/(2\ell)$ keys.

Non-minimal PHF. A difference compared to PtrHash is that in our setting, we do not require a *minimal* PHF. In particular, for PtrHash, fixing (remapping) keys mapping to $[n, n/\alpha)$ is relatively expensive in terms of space, while now we can afford to simply *not* fix them at the cost of a slightly lower load factor. Thus, aside from reducing the space lower bound, non-minimal PHFs allow for more freedom in the design of the data structure.

Bumping. Problematic buckets where none of the 2^8 seeds work are handled differently compared to PtrHash. Specifically, the eviction strategy used in PtrHash has the drawback that it needs to store a lot of metadata to know for each slot the conflicting key, and then also a lot of time is spent repeatedly evicting already placed buckets. Here, we take the much simpler approach of *bumping* problematic buckets, as also done by e.g. PHast [4]. We reserve the seed value 0 for this case, and let it indicate that all keys in the bucket are *bumped*, which means that they will be assigned to a bin using a secondary/fallback data structure. We map the n' bumped keys recursively using the same algorithm, but with lenient parameters of load factor $\alpha' = 0.5$ and double the target number of bits per key. Note that the recursive structure allocates both additional memory for its seeds, and also increases the number of target bins, thereby decreasing the effective load factor α.

At query time, bumped keys are looked up in the fallback structure, and the returned index is added to the number of bins b in the main structure.

Bucket function. With k-perfect hashing, the first keys are easier to place than the later ones. To achieve equal difficulty in each bucket, the keys are distributed non-uniformly. Hermann et al. [29] introduce the optimal bucket function for *minimal* k-PHFs, which is complex to evaluate numerically. Here, we heuristically take $\gamma(x) = x^4$ as a fast-to-compute function that works well despite its simplicity.

An additional benefit of such a skew function is that, for example, half the keys are mapped to the first 6.25% of buckets, so that a tiny fraction of the memory can answer a large fraction of queries, reducing the number of cache misses.

3.3 k-PtrHash: Scoring k-PHF

A drawback of the greedy approach is that it uses the first seed that works, even though there might be multiple seeds that work and some of those other candidates might be "better". Specifically, we want to avoid completely filling up bins to 100% load as long as possible: this gives future keys more flexibility and therefore increases the chances that future buckets find some working seed. Thus, a seed is better if it places the keys into relatively empty bins, and our non-greedy variant chooses the *best* seed out of all $2^8 - 1$ candidates using a scoring heuristic.

Scoring function. Specifically, we introduce a *scoring function* $f : \{0, \dots, k-1\} \to \mathbb{R}$ that assigns a score to adding a key to a bin that currently contains i keys. The score of a seed is the sum of these scores over the target bins of all keys in the bucket, and we choose the seed minimizing this score. When multiple keys map to the same bin, we add $f(i) + f(i+1) + \dots$ rather than multiple times $f(i)$. We heuristically choose the scoring function $f(i) := 2^i$, which means that putting keys in nearly-full bins is heavily penalized compared to using near-empty bins. Experimentation shows that a wide range of functions performs well, and e.g. 1.5^i or 6^i perform almost as good for the parameters we consider ($4 \leq k \leq 16$, $0.7 \leq \alpha \leq 0.99$). We choose $f(i) = 2^i$ for its simplicity.

Engineering the construction algorithm. Like PHast, we use a bitvector to indicate full bins and choose h_2 such that the seeds in $[1, 128)$ and $[128, 256)$ respectively map each key to consecutive bins. This allows us to use bitmasks to quickly filter out seeds that lead to collisions. Further optimizations are explained in Appendix D.

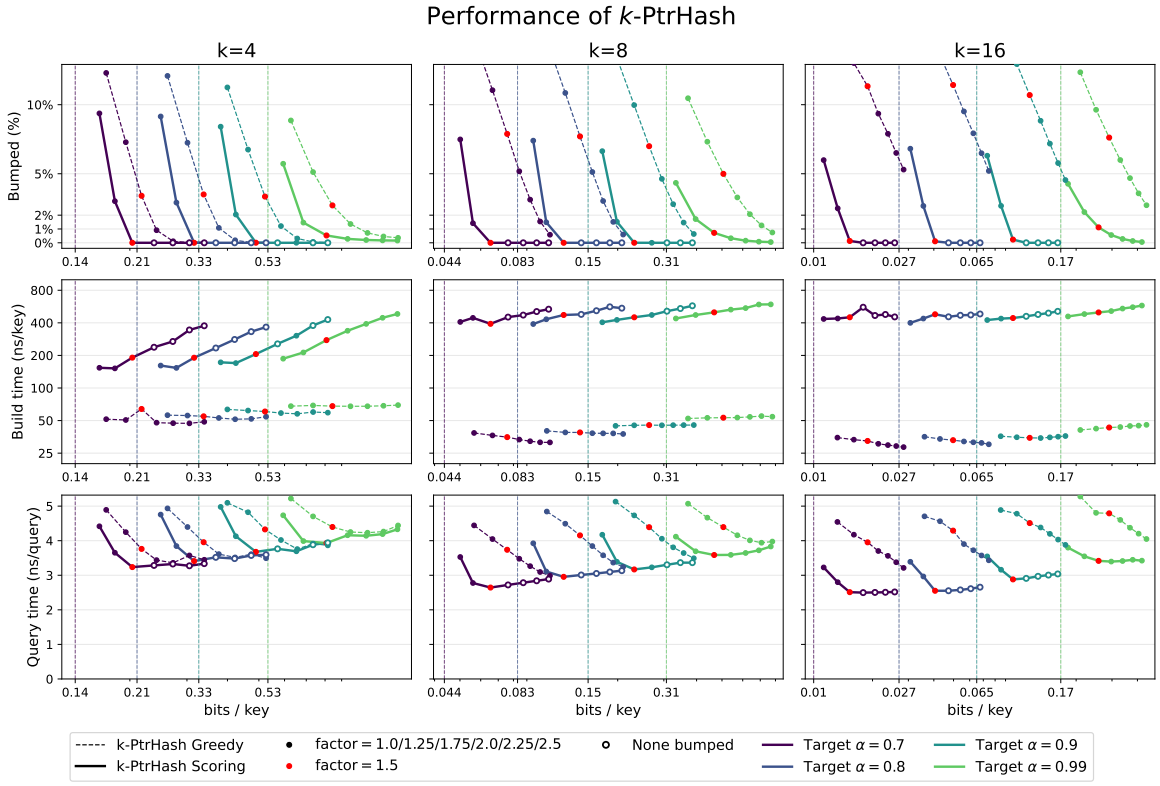

Figure 3 Results of constructing (greedy) k-PtrHash for $n = 10^8$ keys for $k \in \{4, 8, 16\}$. Colours indicate the target load factor $\alpha \in \{0.7, 0.8, 0.9, 0.99\}$. Different points along each line indicate the target space usage relative to the lower bound, with the 1.5× overhead points highlighted in red. The logarithmic x-axis shows the final space usage in bits per key. The top row shows the percentage of bumped keys, with open circles indicating no bumping at all. The scoring variant bumps much less than the greedy variant. The second row shows the (logarithmic) construction time, which is 4× to 10× faster for the greedy variant. Lastly, the bottom row shows the query time, which is faster for smaller structures, but grows again when keys are bumped.

3.4 Results

Our Rust implementation of k-PtrHash is available at github:RagnarGrootKoerkamp/static-hash-sets. Figure 3 shows results of constructing k-PtrHash for various $k \in \{4, 8, 16\}$, $\alpha \in \{0.7, 0.8, 0.9, 0.99\}$, and a target space usage of 1.0× to 2.5× the lower bound (the actual space usage is higher if bumping occurs). Table 3 shows numeric results for the construction with scoring function, corresponding to the data points marked in red that target 1.5× the space lower bound. We see that the scoring variant bumps less than the greedy variant, and can achieve as low as 1.5× space overhead (red dots) with minimal ($< 0.1\%$) bumping for $\alpha \leq 0.9$. This is similar to the compact configuration of PtrHash, which uses 2 bits per key for $k = 1$ and $\alpha = 0.99$, which is 1.46× the lower bound of 1.37 bits per key. We further see that the greedy algorithm is faster during construction as it has fewer seeds to consider. Maybe counterintuitively, the scoring version gets *slower* as the space usage goes up and the problem becomes easier. This is because there are more collision-free candidate seeds whose score has to be computed. As expected, queries are faster when less space is used, but slow down once more than $\approx 1\%$ of the keys are bumped.

k	Load factor α		Space (bits/key)		Bumped	Build time	Query time
	target	final	target	final	(%)	(ns/key)	(ns/key)
4	0.70	≈ 0.700	0.204	0.204	0.001	191	3.2
4	0.80	≈ 0.800	0.316	0.316	0.001	190	3.4
4	0.90	≈ 0.900	0.489	0.490	0.005	206	3.7
4	0.99	0.980	0.800	0.808	0.53	277	3.9
8	0.70	≈ 0.700	0.065	0.065	0.0002	398	2.6
8	0.80	≈ 0.800	0.124	0.124	0.0001	473	3.0
8	0.90	≈ 0.900	0.231	0.231	0.002	450	3.2
8	0.99	0.976	0.460	0.467	0.72	500	3.6
16	0.70	≈ 0.700	0.015	0.015	0.13	450	2.5
16	0.80	0.798	0.040	0.041	0.11	480	2.6
16	0.90	0.896	0.097	0.098	0.23	443	2.9
16	0.99	0.968	0.252	0.258	1.12	499	3.4

Table 3 Results of constructing the scoring variant of k-PtrHash on $n = 10^8$ keys, corresponding to the red highlighted dots in Figure 3 that target 1.5× the space lower bound for each k and load factor. The *final* load factor and space usage take into account the additional slots and space used for bumping. The time for construction is measured with a single thread, and the query time is measured as the average time per iteration of a for loop.

Comparison to previous k-PHFs. Similar to the threshold-based *minimal* k-PHF with consensus developed in [29], we are able to achieve roughly 1.5× the space lower bound for $k \approx 10$. At ≈ 200 ns/key (also for $n = 10^8$), our construction also has a similar construction speed. Although the query performance is not directly comparable due to string hashing, our ≈ 4 ns/query (or 250 Mqueries/s) is much faster than even the fastest threshold-based scheme at ≈ 50 Mqueries/s. The k-RecSplit method gets as low as 1.1× space overhead, but is much slower to build and query.

CHD [2] achieves a space usage of 1.03, 0.77, and 0.60 bits per key for $k \in \{4, 8, 16\}$ respectively when using a load factor of $\alpha = 0.99$. k-PtrHash uses less space at 0.80, 0.47, and 0.25 bits per key respectively, but bumps around 1% of the keys and thus has a smaller overall load factor around 0.98.

Varying k. While the current implementation is tuned towards $4 \leq k \leq 16$, the bumping fallback ensures that construction will always terminate, even for larger k, although not always as space-efficiently. The x^4 bucket function prevents it from working directly for $k = 1$ due to self-collisions in the first (largest) bucket, but changing that to x^2 gives results comparable to PtrHash when α is not too close to 1. For example, for $\alpha = 0.9$ and targeting 1.5× space overhead, we get 2.1% bumped keys and a final load factor of 0.87 at 1.67 bits per key.

4 Application: Static k-PHF-sets

In this section, we will use our non-minimal k-PHF to develop a fast and memory efficient static hash set. We first review some existing (static) hash sets, and then present our own data structure and experiments. For simplicity, we assume 64-bit keys and no associated values. We note that there have been many previous reviews on dynamic hash tables [39, 1, 55].

4.1 Previous work on static hash sets

SwissTable. A commonly used *dynamic* hash map implementation is Google's SwissTable `absl::flat_hash_map` [24], which is also the basis for Rust's standard `HashMap`, which is based on the `hashbrown` crate. It stores an array of n/α slots with load factor $7/16 \leq \alpha \leq 7/8$, and additionally a metadata array of 7-bit *fingerprints* (and 1 tombstone bit marking deletions) for each slot. A hash function is used to map a query key to a group of 8 slots (or more, when using an implementation with SIMD). Its fingerprint is then compared, and quadratic probing to the next group happens until either a matching fingerprint or an empty slot is found. Once a fingerprint matches, the query is checked against the corresponding key.

The fingerprints act as a small (and thus fast) approximate filter for negative queries. On average, $\Omega(1/\epsilon)$ probes are needed per query (where $\epsilon = 1 - \alpha$) before the key or an empty/deleted slot is found. Efficient prefetching is possible if and insofar as the first probe is sufficient for answering a query A drawback is that the table only supports sizes that are powers of two, causing worst-case 2.8× space overhead compared to just an array of keys.

Static cache line-based hash set. A simple implementation of a static hash set is the following: Allocate $\lceil n/\alpha/8 \rceil$ 512-bit cache line sized bins that each fit 8 64-bit keys. Then hash each key to a cache line, and use linear probing until a bin is found that still has capacity, and insert it there. Empty slots are represented by 0, and 0 itself is handled separately by simply storing a boolean indicating whether 0 is present or not. To query a key, the key is compared (using SIMD instructions) against all values in a cache line, with linear probing until either the key or an empty slot is found.

This has the same drawbacks of probing as SwissTable does, but avoids the indirection to the fingerprint metadata. This is especially beneficial in cases with little probing, i.e., for positive queries when the load factor is small.

Blocked cuckoo hashing. Blocked Cuckoo-hashing [46, 18, 15] avoids unbounded probing by selecting exactly two candidate cache line sized blocks (or windows [51, 48]) for each key. When both blocks are already full, one of the current keys in them is evicted and a BFS or DFS is done to displace them. To query whether a key is present, one can either *eagerly* load both cache lines from memory at the same time (doubling the required memory bandwidth) or *lazily* load the second only when the key is not in the first (increasing latency in case a key is not present in its primary location).

1-**PHF-set.** Instead of probing, one can also use a 1-PHF to map keys to slots without collisions [49, 41]. FPH [54] is a fast implementation that uses a non-minimal 1-PHF based on FCH [19]. This PHF directly maps each key to a single slot, which stores the value of the key mapping there. As long as the PHF fits in cache, this allows for fast single-cache-miss queries, with no performance distinction between positive and negative queries. The same is possible by using PtrHash or PHast, which have less space overhead and higher query throughput than FCH.

Previous k-PHF approaches. The cache line-based approach and 1-PHF approach can be merged by using a k-PHF to map keys to cache lines, where k is the number of keys per cache line. For example one can use the non-minimal k-PHF CHD [2] or the *minimal* k-PHFs developed in [29]. Another line of work considers dynamic hash tables that guarantee a single cache miss and thus implicitly contain a larger *dynamic* k-PHF. Notable techniques are separator hashing [23, 33, 32], MapEmbed on CPU/FPGA [53], external Robin-Hood hashing [9], EEPH for persistent memory [12], or GPH for GPUs [8]. We remark that the extendible hashing technique [17, 45] has up to two cache misses.

4.2 The static k-PHF-set

We extend the 1-PHF-sets based on 1-PHF to a static k-PHF-set based on k-PtrHash. This enables a high-load-factor hash set that only needs to read exactly one cache line from RAM as long as the k-PHF fits in memory. As in the previous methods, we treat each cache line as a bin with $k = 8$ slots, and use SIMD instructions to check if a key is contained in a cache line. The main benefit is that a static k-PHF needs much less space than a static 1-PHF necessarily needs, and much less than a dynamic k-PHF needs in practice, allowing a fixed-size cache to store a PHF for a larger number of keys. Starting with minimal 1-PHFs which needs 1.44 bits/key, going to $\alpha = 0.9$ requires 1.07 bits/key, and then increasing k to 8 lowers the space lower bound to 0.15 bits/key, 7× less than for $k = 1$. Thus, a smaller α makes the k-PHF-set more cache-friendly and faster, even though more memory is used.

4.3 Results

Competitors. We compare our k-PHF-set based on k-PtrHash against the methods mentioned in Section 4.1. The first is the Rust `HashSet` in `hashbrown`. We implemented the blocked cuckoo hashing ourselves as a known baseline using the same SIMD instructions as the version based on k-PtrHash. (We omit the linear-probing variant as it is consistently slower.) We call `FphDynSet` [54] through Rust bindings and also added support for prefetching the slot of a key. We wrap the fastest configurations of PHast+ and PtrHash in our own 1-PHF-set. All Rust implementations use GxHash [21] as the primary hash function.

We also tested the k-PHF-set with the fastest bucket placement and threshold based bumping minimal k-PHF techniques of [29], but both were ≈ 2 times slower than the standard Rust HashSet. Similarly, MapEmbed [53] was ≈ 5 times slower, while we could not find any source code for separator hashing [23, 33, 32], external Robin-Hood hashing [9], extendible hashing [17] and EEPH [12]. The query time of the k-PHF technique CHD [2] is slower compared to PtrHash by an order of magnitude. Thus, these methods were excluded.

For each static method, we show the best load factor. For the linear probing based hash set and the lazy cuckoo set, we use $\alpha = 0.7$ to reduce probing costs. For the eager cuckoo set, we use $\alpha = 0.99$ since we always read both locations anyway. While FPH works for $\alpha \leq 0.98$, we choose $\alpha = 0.9$ to avoid excessively slow construction on large inputs. PHast+ and PtrHash without remapping naturally have a load factor around 0.98 and 0.99 respectively. For k-PHF-Set with k-PtrHash, we test load factors $\alpha \in \{0.7, 0.9, 0.95\}$.

Setup. As our data structure is designed to be I/O-efficient, we benchmark it in a *high throughput* setting: we do 3 million queries, and *prefetch* the cache line for the query 32 iterations ahead. Specifically, the 1-PHF-set and k-PHF-set compute the index of the slot/bin a key maps to and prefetch this cache line. This index is then stored, and the corresponding data is read 32 iterations later to test whether the key is indeed present, so that the key does not have to be hashed again. We forked `hashbrown` and FPH to add similar support for prefetching and continuing the query some time later. For cuckoo hashing, we test two modes: one prefetching only the first (preferred) cache line, and one where we eagerly fetch the second location as well.

Additionally, we test a mock implementation of a basic hash set based on cache lines. This maps each random 64-bit key to one of the b cache lines via a single multiplication using $\lfloor k \cdot b/2^{64} \rfloor$ and then "checks" if the key is in the hash set by testing if it equals the first word in the cache line. This indicates the maximum achievable throughput of a hash set that does not use an internal filter.

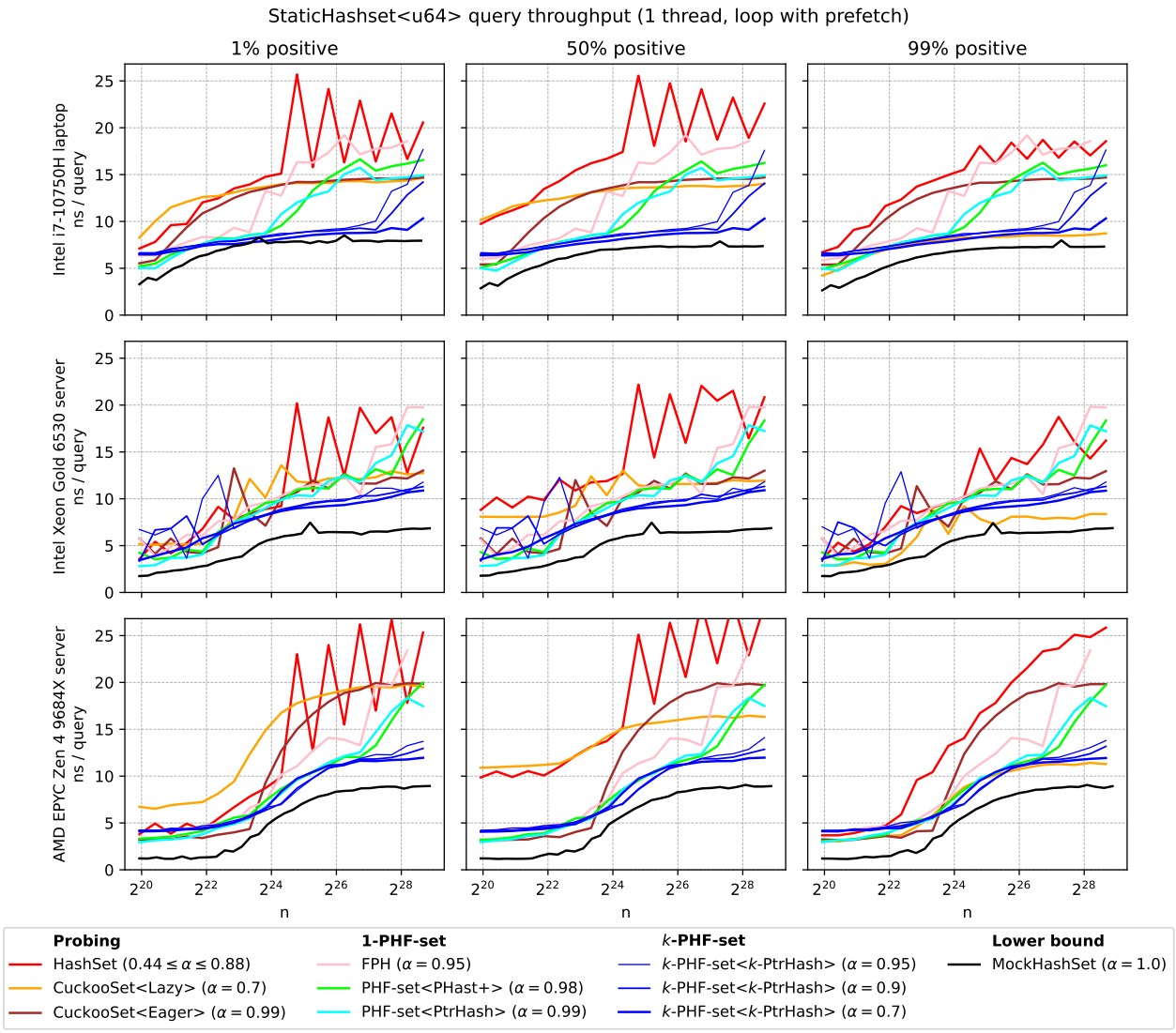

Figure 4 Query throughput of various (static) hash set implementations for n 64-bit keys. RAM accesses are prefetched 32 loop iterations ahead. Colours indicate different data structures, and for k-PHF-set, the line width indicates the load factor from $\alpha = 0.7$ (thick) to $\alpha = 0.95$ (thin). The black line indicates the throughput of a mock implementation that checks if the key equals the first word in a random cache line, and thus gives an indication of the maximum achievable throughput of a hash set that does not use an internal filter. The left/middle/right column show throughput for 1% / 50% / 99% positive queries. The top/middle/bottom are for an Intel laptop, Intel Xeon server, and AMD EPYC server. Appendix E contains further results on multiple threads and without prefetching.

We run the benchmarks on an Intel Skylake i7-10750H laptop (6 cores, 32 KiB L1 per core, 256 KiB L2 per core, 12 MiB L3 cache), an Intel Xeon Gold 6530 server (64 cores, 128 threads, 32 KiB L1 per core, 2 MiB L2 per core, 160 MiB L3 cache per 32 cores), and an AMD EPYC Zen 4 9684X server (96 cores, 192 threads, 64 KiB L1 per core, 1 MiB L2 per core, 32+64 MiB L3 per 8 cores).

Appendix E contains additional results without prefetching and with multithreading.

Discussion. Results are shown in Figure 4. The dynamic `HashSet` (red) is relatively slow, and its speed depends heavily on the load factor, which varies by n due to the power-of-2 restriction on the underlying buffer. (We note that in the additional results in Appendix E there are cases where it is the fastest for negative queries, due to the 7-bit filter. Indeed, this allows it to be faster than the lower bound given by our mock implementation.) The lazy cuckoo table (orange) is fast for positive queries since (due to the small load factor) these require nearly no probing, but in other cases it is as slow as the eager variant (brown) that has double the memory bandwidth. The three 1-PHF-sets (pink, lime, cyan) all perform similar and independent of the type of queries, with FPH being slightly slower than using PHast+ or PtrHash. Specifically, they remain fast up to $n \approx 2^{24}$, whereas the probing methods slow down at $n \approx 2^{21}$.

As anticipated, using k-PtrHash indeed allows larger n up to 2^{27} (on the laptop) before the PHF does not fit in cache anymore and queries slow down, compared to $n = 2^{24}$ for the 1-PHF-set, resulting in up to 1.6× higher throughput in this range. Decreasing α from 0.95 to 0.7 maintains high throughput up to 2^{29}, as the k-PHF takes less space for smaller α.

The two server machines have larger caches, and our k-PHF-set only starts to be faster than the 1-PHF-set at $n = 2^{27}$ here. Due to the different memory characteristics of these machines, the cuckoo table is generally competitive on the Intel Xeon machine, but slow on the AMD EPYC, where the k-PHF-set has up to 1.5× higher throughput than other methods for negative queries. Overall, our k-PHF-set is the only method that is competitive on all machines for both positive and negative queries.

5 Conclusion

We have given a tight lower bound on the space usage of k-PHFs with load factor $0 < \alpha < 1$ (Theorem 8). As the formula for the exact space usage is somewhat unwieldy, we also derive explicit simpler formulas that neglect constant factors. We extended PtrHash to k-perfect hashing with two modifications: we bump problematic buckets rather than using evictions, and we use a scoring function to preferentially select seeds that result in more balanced bins. For $4 \leq k \leq 16$ and $0.7 \leq \alpha \leq 0.9$, this scheme works down to 50% space overhead.

An 8-PHF-based hash set that maps 64-bit keys to cache line-sized bins has as high or higher throughput than other hash sets, and depending on the hardware has up to 1.5× higher throughput in cases where a 1-PHF on the data does not fit in cache.

Future work. It appears that none of the current highly space-efficient strategies for constructing PHFs generalize well to the non-minimal k-PHF setting, and it remains open to find methods that are close to the space lower bound.

Concerning k-PtrHash, much remains unclear on e.g. the optimal combination of bucket function and scoring function to use. From the practical side, more work could be done to speed up the k-PtrHash construction using e.g. SIMD instructions, and to extend k-PHF-set into a map with updatable values. Additionally, it might be possible to use quotienting to store only a subset of the bits of each key, thus allowing a larger k.

References

1 Jackson Allan. An Extensive Benchmark of C and C++ Hash Tables. https://jacksonallan.github.io/c_cpp_hash_tables_benchmark/, 2024.

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

3 Piotr Beling. Fingerprinting-based minimal perfect hashing revisited. *ACM J. Exp. Algorithmics*, 28:1.4:1–1.4:16, 2023. doi:10.1145/3596453.

4 Piotr Beling and Peter Sanders. Phast – perfect hashing made fast. In *ALENEX*, pages 1–14, 2026. doi:10.1137/1.9781611978957.1.

5 Dominik Bez et al. High performance construction of RecSplit based minimal perfect hash functions. In *ESA*, volume 274 of *LIPIcs*, pages 19:1–19:16, 2023. doi:10.4230/LIPICS.ESA.2023.19.

6 Fabiano C Botelho, David M Gomes, and Nivio Ziviani. A new algorithm for constructing minimal perfect hash functions. *Preprint*, 2004.

7 Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *CIKM*, pages 653–662. ACM, 2007. doi:10.1145/1321440.1321532.

8 Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proceedings of the ACM on Management of Data*, 3(3):1–26, June 2025. doi:10.1145/3725406.

9 Pedro Celis. *External robin hood hashing*. PhD thesis, University of Waterloo, 1988.

10 Jarrod A. Chapman et al. Meraculous: De novo genome assembly with short paired-end reads. *PLOS ONE*, 6(8):1–13, 08 2011. doi:10.1371/journal.pone.0023501.

11 Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39. SIAM, 2004.

12 Qi Chen, Hao Hu, Cai Deng, Dingbang Liu, Shiyi Li, Bo Tang, Ting Yao, and Wen Xia. EEPH: An Efficient Extendible Perfect Hashing for Hybrid PMem-DRAM. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, page 1366–1378. IEEE, April 2023. doi:10.1109/icde55515.2023.00109.

13 Bede Constantinides, John Lees, and Derrick W Crook. Deacon: fast sequence filtering and contaminant depletion. *bioRxiv*, 2025. doi:10.1101/2025.06.09.658732.

14 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.

15 Martin Dietzfelbinger et al. Tight thresholds for cuckoo hashing via XORSAT. In *ICALP*, volume 6198 of *LNCS*, pages 213–225. Springer, 2010. doi:10.1007/978-3-642-14165-2_19.

16 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.

17 Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979. doi:10.1145/320083.320092.

18 Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. *Comb. Probab. Comput.*, 25(6):870–908, 2016. doi:10.1017/S0963548315000334.

19 Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *SIGIR*. ACM Press, 1992. doi:10.1145/133160.133209.

20 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *SEA*, volume 9685 of *LNCS*, pages 339–352. Springer, 2016. doi:10.1007/978-3-319-38851-9_23.

21 Olivier Giniaux. Gxhash: A high-throughput, non-cryptographic hashing algorithm leveraging modern CPU capabilities, 2023. doi:10.5281/ZENODO.8368254.

22 B. V. Gnedenko. On a local limit theorem of the theory of probability. *Uspekhi Mat. Nauk*, 3:187–194, 1948. URL: http://mi.mathnet.ru/eng/rm8713.

23 Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *J. ACM*, 35(1):161–184, 1988. doi:10.1145/42267.42274.

24 Google. Abseil's Swiss Table. https://abseil.io/about/design/swisstables, 2017.

25 Ragnar Groot Koerkamp. PtrHash: Minimal Perfect Hashing at RAM Throughput. In *SEA*, volume 338 of *LIPIcs*, pages 21:1–21:21, 2025. doi:10.4230/LIPIcs.SEA.2025.21.

26 Ragnar Groot Koerkamp, Stefan Hermann, Peter Sanders, and Stefan Walzer. Non-minimal k-perfect hashing: Tight lower bounds and an application to fast static hash tables. *arXiv*, 2026. doi:10.48550/ARXIV.2607.07257.

27 Stefan Hermann. Morphishash: Improving space efficiency of shockhash for minimal perfect hashing. *LIPIcs, Volume 351, ESA 2025*, 351:9:1–9:16, 2025. doi:10.4230/LIPICS.ESA.2025.9.

28 Stefan Hermann et al. PHOBIC: perfect hashing with optimized bucket sizes and interleaved coding. In *ESA*, volume 308 of *LIPIcs*, pages 69:1–69:17, 2024. doi:10.4230/LIPICS.ESA.2024.69.

29 Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Engineering minimal k-perfect hash functions. In *ESA*, volume 351, pages 99:1–99:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ESA.2025.99.

30 Kumar Joag-Dev and Frank Proschan. Negative association of random variables with applications. *The Annals of Statistics*, 11(1):286 – 295, 1983. doi:10.1214/aos/1176346079.

31 Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. PaCHash: Packed and compressed hash tables. In *ALENEX*, pages 162–175. SIAM, 2023. doi:10.1137/1.9781611977561.CH14.

32 Per-Åke Larson. Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval. *ACM Trans. Database Syst.*, 13(3):366–388, 1988. doi:10.1145/44498.44500.

33 Per-Åke Larson and Ajay Kajla. File organization: Implementation of a method guaranteeing retrieval in one access. *Commun. ACM*, 27(7):670–677, 1984. doi:10.1145/358105.358193.

34 Hans-Peter Lehmann. *Fast and Space-Efficient Perfect Hashing*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2024. doi:10.5445/IR/1000176432.

35 Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. Modern minimal perfect hashing: A survey. *ACM Computing Surveys*, 58(10):1–36, March 2026. doi:10.1145/3797036.

36 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.

37 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force. In *ALENEX*, pages 194–206. SIAM, 2024. doi:10.1137/1.9781611977929.15.

38 Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler. Combined search and encoding for seeds, with an application to minimal perfect hashing. *LIPIcs, Volume 351, ESA 2025*, 351:109:1–109:18, 2025. doi:10.4230/LIPICS.ESA.2025.109.

39 Martin Leitner-Ankerl. Comprehensive C++ Hashmap Benchmarks 2022. https://martin.ankerl.com/2022/08/27/hashmap-bench-01/, 2022.

40 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, 2017. doi:10.4230/LIPICS.SEA.2017.25.

41 Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, page 2774–2778. IEEE, July 2006. doi:10.1109/isit.2006.261567.

42 Harry G. Mairson. The program complexity of searching a table. In *FOCS*, pages 40–47. IEEE, 1983. doi:10.1109/SFCS.1983.76.

43 Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996. doi:10.1093/COMJNL/39.6.547.

44 Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *FOCS*, pages 170–175. IEEE, 1982. doi:10.1109/SFCS.1982.80.

45 Rasmus Pagh. Basic external memory data structures. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 14–35. Springer, 2003. doi:10.1007/3-540-36574-5_2.

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

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

48 Reiner Pope. Cuckoo hashing improves SIMD hash tables (and other hash table tradeoffs). https://reiner.org/cuckoo-hashing, 2025.

49 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.

50 Zbigniew Szewczak and Michel Weber. Classical and almost sure local limit theorems. *arXiv*, 2022. doi:10.48550/ARXIV.2208.02700.

51 Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. *ACM Transactions on Algorithms*, 19(3):1–22, May 2023. URL: http://dx.doi.org/10.1145/3589558, doi:10.1145/3589558.

52 Sean A. Weaver and Marijn Heule. Constructing minimal perfect hash functions using SAT technology. In *AAAI*, pages 1668–1675. AAAI Press, 2020. doi:10.1609/AAAI.V34I02.5529.

53 Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. Mapembed: Perfect hashing with high load factor and fast update. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 1863–1872. ACM, August 2021. doi:10.1145/3447548.3467240.

54 Ren Zibei. Flash Perfect Hash. https://github.com/renzibei/fph-table, 2022.

55 Ren Zibei. Hash Table Benchmark. https://github.com/renzibei/hashtable-bench, 2022.

A Simple Brute Force Suffices for Large Universes

We fill in the missing proof of Theorem 7, meaning for large universes a brute force algorithm need not enforce a balancing condition on the random functions it tries.

▶ **Theorem 7.** *Assume $u \geq n^2/\alpha$. Then simple brute force is space optimal in the same sense as balanced brute force (see Theorem 6).*

Proof. It suffices to show that the probability q_{simp} that a uniformly random function is compatible with a given input set satisfies $q_{\text{simp}} = \Theta(q_{\text{bal}})$, as taking logarithms will turn this into an additive $+\mathcal{O}(1)$ discrepancy.

Let $\mathcal{G}$ be the set of all k-perfect assignments $g : S \to [b]$. Moreover, for $g \in \mathcal{G}$ let $p_{\text{simp}}(g)$ (respectively $p_{\text{bal}}(g)$) be the probabilities that a uniformly random (balanced) function matches g when restricted to S. We have $q_{\text{simp}} = \sum_{g\in\mathcal{G}} p_{\text{simp}}(g)$ and $q_{\text{bal}} = \sum_{g\in\mathcal{G}} p_{\text{bal}}(g)$ so it suffices if we show that $p_{\text{simp}}(g) = \Theta(p_{\text{bal}}(g))$ for all $g \in \mathcal{G}$. We immediately see $p_{\text{simp}}(g) = b^{-n}$, which means we have to show $p_{\text{bal}}(g) = b^{-n} \cdot \Theta(1)$.

To get at $p_{\text{bal}}(g)$, we use the chain rule. The probability that a random balanced function maps the ith key $x_i \in S$ according to g *conditioned* on having $x_1, \ldots, x_{i-1} \in S$ according to g is of the form $\frac{c}{d}$ where $c \in [\frac{u}{n} - k, \frac{u}{b}]$ is the remaining number of occurrences of $g(x_i)$ and $d \in [u - n, u]$ is the total number of unassigned function values. We can bound $\frac{c}{d}$ using the assumption $u \geq n^2/\alpha$

$$b^{-1}e^{-\Theta(1/n)} \leq b^{-1}(1-\tfrac{n}{\alpha u}) = b^{-1}(1-\tfrac{kb}{u}) = \tfrac{u/b-k}{u} \leq \tfrac{c}{d} \leq \tfrac{u/b}{u-n} \leq b^{-1}(1+\tfrac{n}{u-n}) \leq b^{-1}e^{\Theta(1/n)}.$$

This implies $\frac{c}{d} = b^{-1} \cdot e^{\pm\Theta(1/n)}$ and n such factors as we collect them with the chain rule multiply to $b^{-n}\Theta(1)$ as claimed. ◀

B Numerical approximations of $S(\alpha, k)$

The following sagemath script (see it in action here) computes the value $S(\alpha, k)$ as defined in Theorem 8. It is optimised for readability and faithfulness to Theorem 8 and will, in this form fail if k is too large or α too close to 1.

```
k = 8
α = 0.8
ζ(λ) = 1/sum([λ**i/factorial(i) for i in range(k+1)])
𝔼Z(λ) = sum([i*ζ(λ)*λ**i/factorial(i) for i in range(k+1)])
λ = find_root(𝔼Z(x) == α*k, 0, 1000000)
space = log(λ*e/(α*k),2) + log(ζ(λ),2)/(α*k)
print(N(space)) // prints 0.0829450615181708
```

C Proof of Theorem 10

We prove the following theorem from Section 2.3.

▶ **Theorem 10.** *Let $k \in \mathbb{N}$, $\alpha \in [\frac{1}{2}, 1]$, $s := \lfloor(1-\alpha)k+1\rfloor$ and $n \geq n_0(k, \alpha)$ sufficiently large. Assume $u \geq n^2/\alpha$. The best possible space requirement for k-PHFs with load factor α is*

for $s = \Theta(\sqrt{k})$:	$\Theta(1)$ *bits per bin*	*(total of $\Theta(b) = \Theta(\frac{n}{\alpha k})$ bits),*
for $s \ll \sqrt{k}$:	$\Theta(\log(\sqrt{k}/s))$ *bits per bin*	*(total of $\Theta(b\log(\sqrt{k}/s))$ bits),*
for $s \gg \sqrt{k}$:	$\exp(-\Theta(s^2/k))$ *bits per bin*	*(total of $\Theta(b\exp(-\Theta(s^2/k)))$ bits).*

Recall the roles of $(X_1, \dots, X_b)$ and q_{simp}. We make a few observations before starting the main proof.

▶ **Lemma 12.** *Theorem 10 is true for $k = \Theta(1)$.*

Proof. If $k = \Theta(1)$ then we are necessarily in the case of medium load where a space requirement of $\Theta(b) = \Theta(n)$ is claimed. This is clearly correct:

- $\log_2(e)n = \Theta(n)$ bits are sufficient even in the hardest case of 1-MPHF (a k-MPHF can be obtained by constructing a 1-MPHF and composing it with $i \mapsto \lceil i/k \rceil$).
- $\Theta(1)$ bits per bin are necessary as $q_{\text{simp}} \le \Pr[X_1 \le k]^b$ by Lemma 11 and $\Pr[X_1 \le k] = 1 - \Omega(1)$. ◀

We will never explicitly appeal to Lemma 12. However, keeping in mind that k is large, we allow ourselves to be a bit sloppy with rounding issues and we may appeal to the intuition that the X_i approximately follow a (discretised) normal distribution. This should make the validity of the following lemma apparent without proof.

▶ **Lemma 13.** *Let $N \in \mathbb{N}$, $p \in (0, \frac{1}{2}]$ and $X \sim \text{Bin}(N, p)$. Assume $\mu := Np = \Theta(k)$. Then:*

- *We have $\sigma^2 = \text{Var}(X) = Np(1-p) = \Theta(k)$.*
- *For any $i \in \mathbb{N}$ with $|i - \mu| = \mathcal{O}(\sqrt{k})$ we have $\Pr[X = i] = \Theta(1/\sqrt{k})$.*
- $\max_{i \in \mathbb{N}} \Pr[X = i] = \Theta(\Pr[X = \lfloor \mu \rfloor]) = \Theta(1/\sqrt{k})$.

We will require the following Chernoff-type result:

▶ **Lemma 14.** *Let $N \in \mathbb{N}$, $p < \frac{1}{3}$ and $X \sim \text{Bin}(N, p)$. Then $\Pr[X \ge N/2] = \exp(-\Theta(N \log \frac{1}{p}))$.*

Proof. A standard theorem due to Chernoff and Hoeffding guarantees that

$$\Pr[X \ge N(p + \varepsilon)] \le \exp(-D(p + \varepsilon \| p)n)$$

where $D(x\|y) = x \ln \frac{x}{y} + (1-x) \ln \frac{1-x}{1-y}$ is known as the Kullback-Leibler divergence. Plugging in $\varepsilon = \frac{1}{2} - p$ yields the stated result. ◀

We now have all the tools required for proving Theorem 10. We consider six cases separately.

Lower bound for a case of medium load ($s = \Theta(\sqrt{k})$). By Lemma 14 we have $\Pr[X_1 > k] = \mathcal{O}(1)$ so the lower bound of $\Theta(b)$ bits follows from Lemma 11 just like in the proof of Lemma 12.

Upper bound for a case of medium load ($s = \Theta(\sqrt{k})$). We can handle this case together with the heavily loaded case (considered next). If $s > \sqrt{k}/2$ we may add phantom keys until $s = \sqrt{k}/2$. The resulting upper bound will be $\mathcal{O}(b \log(\sqrt{k}/s))$, which simplifies to $\mathcal{O}(b)$.

Upper bound for the heavily loaded case ($s \le \sqrt{k}/2$). We will show $q_{\text{simp}} = \Pr[\max_{i \in [b]} X_i \le k] \ge (\Omega(s/\sqrt{k}))^b$, which implies $\log_2 \frac{1}{q_{\text{simp}}} = \mathcal{O}(b \log(\sqrt{k}/s))$ as desired. In fact, we will show the stronger claim that with $T_i := X_1 + \ldots + X_i - \mathbb{E}[X_1 + \ldots + X_i]$ for $i \in [b]$ we have

$$\Pr[\forall i \in [b] : X_i \le k \wedge T_i \ge 0] = (\Omega(s/\sqrt{k}))^b. \tag{5}$$

Intuitively, we require that when we reveal $X_1, X_2, \dots, X_b$ one by one, then in step i, we fail if $X_i > k$, but we also fail if we "fall behind" the number of keys that are expected in the first i bins. Consider the success probability of the ith step:

$$\begin{aligned}\Pr[X_i \le k \wedge T_i \ge 0 \mid \forall j < i : X_j \le k \wedge T_j \ge 0] &\ge \Pr[X_i \le k \wedge T_i \ge 0 \mid T_{i-1} = 0] \\ &= \Pr[X_i \in [\alpha k, k] \mid T_{i-1} = 0] = \Omega(s/\sqrt{k}).\end{aligned}$$

In the first step, we remove irrelevant information from the conditioning and switch to the "hardest" case with $T_{i-1} = 0$ (this ignores a rounding issue) where we have "barely not fallen behind". The second step uses $T_i = T_{i-1} + X_i - \alpha k$. Finally we apply Lemma 13 as the conditional distribution of X_i is $\mathrm{Bin}(n - \mathbb{E}[X_1 + \cdots + X_{i-1}], \frac{1}{b-i+1}) = \mathrm{Bin}(n \cdot \frac{b-i+1}{b}, \frac{1}{b-i+1})$ which has expectation $\frac{n}{b} = \alpha k = \Theta(k)$. Now Equation (5) follows from the chain rule.

Lower bound for the heavily loaded case ($s \ll \sqrt{k}$). Call a bin $i \in [b]$ *good* if $X_i \in [k-2s, k]$ and let G be the number of good bins. We begin by showing that the event $\max_{i\in[b]} X_i \le k$ implies $G \ge b/2$.

Assume, conversely, that $\max_{i\in[b]} X_i \le k$ and $G < b/2$. The non-good bins must all have load less than $k - 2s$, the others load at most k. The total average load is hence less than $\frac{1}{2}(k-2s) + \frac{1}{2}k \le k - s = \alpha k$. This is a contradiction as the average load is equal to αk.

We now again reveal $X_1, X_2, \ldots, X_b$ one by one. In step $1 \le i < b$ the conditional distribution of X_i is binomial and the chance that bin i is good is maximised if the conditional expectation of X_i is around $k-s$. We hence have $\Pr[\text{bin } i \text{ is good} \mid X_1, \ldots, X_{i-1}] = \mathcal{O}(s/\sqrt{k})$ by Lemma 13. A standard coupling argument helps to construct a random variable $G' \sim \mathrm{Bin}(b, p)$ with $p = \mathcal{O}(s/\sqrt{k})$ such that $G \le G'$. Note that the bound on p implies $p < \frac{1}{3}$. We conclude

$$q_{\text{simp}} = \Pr[\max X_i \le k] \le \Pr[G \ge \tfrac{b}{2}] \le \Pr[G' \ge \tfrac{b}{2}] \overset{\text{Lem.14}}{=} \exp(-\Omega(b \log \tfrac{1}{p})).$$

Hence $\log_2 \frac{1}{q_{\text{simp}}} = \Omega(b \log \frac{1}{p}) = \Omega(b \log(\sqrt{k}/s))$ as required.

Upper bound for the lightly loaded case ($s \gg \sqrt{k}$). Note that every bin has $s/\sqrt{k}$ many standard deviations worth of extra space compared to its average load. The probability that a bin overflows is hence $p = \Pr[X_i > k] = \exp(-\Omega(s^2/k))$ by a standard Chernoff bound.

We will build a k-PHF by using a random assignment and fixing these overflowing bins. Firstly, we store the set of overflowing bins using entropy coding. This takes $bp \log \frac{1}{p} + (1-p) \log \frac{1}{1-p} = \Theta(bp \log \frac{1}{p}) = \mathcal{O}(b\sqrt{p}) = b \exp(-\Omega(s^2/k))$ bits in expectation, which is within our budget. We now describe how to redistribute keys from overflowing bins to other bins using, in expectation, $\mathcal{O}(1)$ bits per overflowing bin. Concretely we store for each overflowing bin:

1. The number $D = \lceil (X_i - k)/\sqrt{k} \rceil$ of standard deviations by which it overflows. Note that $\mathbb{E}[D] = \mathcal{O}(1)$, even conditioned on $X_i > k$. Hence even unary encoding of D is sufficient.
2. A sequence of up to $2D$ seed values (in unary encoding) of hash functions that partition the X_i keys, each hash function splitting off between $\sqrt{k}/2$ and $\sqrt{k}$ keys. We expect each seed to take $\mathcal{O}(1)$ bits as we have a constant success probability. The up to $2D$ resulting subsets are called overflow fragments.
3. Imagine each overflow fragment randomly probes the set of all bins until finding a bin with $\sqrt{k}$ spare slots. We store, in unary encoding, the index of the successful probe. As a constant fraction of bins has $\sqrt{k}$ free slots in expectation, and we expect $o(b)$ overflow fragments (start from scratch if this fails), this also takes $\mathcal{O}(1)$ bits in expectation.

While not elegant, it should be clear that the resulting data structure can serve as a k-PHF.

Lower bound for the lightly loaded case ($s \gg \sqrt{k}$). We use that $\Pr[X_i > k] = \exp(-\Theta(s^2/k))$, which follows from local limit theorems[4] assuming $n = \omega(k^6)$. We ap-

[4] See e.g. lecture notes by Steven Dunbar (see "key concepts" → 2).

ply Lemma 11 to get

$$q_{\text{simp}} = \Pr[\max X_i \leq k] \leq \Pr[X_1 \leq k]^b = (1 - \exp(-\Theta(s^2/k)))^b.$$

Using that $\log \frac{1}{1-\varepsilon} = \Theta(\varepsilon)$ for small ε we get as desired

$$\log_2 \frac{1}{q_{\text{simp}}} \geq b \log \frac{1}{1 - \exp(-\Theta(s^2/k))} = b \exp(-\Theta(s^2/k)).$$

D Engineering the construction algorithm

Our implementation makes a few specific choices to ensure that construction is fast.

Shifting bucket function. For each seed that is tried for a bucket, we must compute $\text{slot}(k_i) = h_2(k_i, \text{seed})$ for all keys in the bucket. If h_2 is a truly random function, that means we have a memory access and likely cache miss for each key for each seed in order to check whether the bin is already full. In order to reduce this, we only use the high bit of the 8-bit seed for a random hash position, and the low 7 bits indicate a linear shift relative to that position:

$$\text{slot}(k_i) = h_2(k_i, \text{seed}) := h_2'(k_i, \text{seed} \,\&\, 1000\,0000_2) + (\text{seed} \,\&\, 0111\,1111_2). \tag{6}$$

This way, seeds 1 to 127 and 128 to 255 map each key to bins that are adjacent in memory.

It is also possible to simply use *all* 8 seed bits to indicate a shift. While this works mostly fine in practice, it increases the probability of self-collisions in the first bucket (see below), and usually causes slightly (but occasionally 2×) more bumping. Using the high bit for the position effectively enables a windowed cuckoo-hashing-like effect [51], mitigating cases where many keys map into the same range of bins.

Bitmasks. Similar to PHast [4], we further speed things up by keeping a bitmask that indicates which bins still have space. When processing seeds 0 and 128, we read 128 bits from the mask starting at the bin for each key, and then intersect these masks. Then, for each of the next 128 seeds we first check whether the corresponding bit is 1 before processing it further.

Self-collision checks. After the bitmask check passes, there can still be collisions when two keys in a bucket map to a bin with only one free slot. To efficiently handle this for consecutive seeds, we compute for seeds 0 and 128 a sorted list of pairs $(bin, count)$ indicating how many keys map to each bin. For the remaining seeds we simply increment the bin index as needed. For each seed, we then iterate over the list and simply check whether each bin has sufficient capacity remaining.

Computing the score. To compute the score, we keep a list $C = [0, \ldots, 0]$ of $k+1$ counts. When adding c keys to a bin with size i, we update $C_i \leftarrow C_i + 1$ and $C_{i+c} \leftarrow C_{i+c} - 1$. After taking prefix sums, C_i then indicates the number of times a key was added to a bin with i keys, and we can compute the final score as the dot product $\sum_{i=0}^{k-1} C_i \cdot f(i)$.

Global seed. Typically, the first bucket receives up to 20% of the keys. Because of this, the probability that for any fixed seed there is some bin receiving at least $k+1$ of these keys is not negligible. Due to the shifting based on the seed, only 2 independent hash functions are tried, and it can happen (empirically around 10% of the time) that both lead to self-collisions.

We work around this by making h_1 dependent on a random global seed, so that we can retry construction until the first bucket does not lead to overflowing bins.

This global seed also serves a secondary purpose to randomize the hashes of keys, so that the fallback structures for bumped keys have an independent hash function.

D.1 Consensus

An alternative way to handle problematic buckets that we considered (but do not use) is *consensus* [38]: we can use as hash a 64-bit seed ending in the 8-bit seed of the current bucket. This way, the seed of preceding buckets affects the placement of the seeds in the current bucket, and we can backtrack if no suitable placement is possible.

While this avoids the need for bumping and the associated cost at query time, preliminary experiments showed that bumping is more effective at resolving issues and achieves smaller space overheads. Furthermore, the metadata required for backtracking somewhat complicates construction, and reading 64-bit seeds across cache line boundaries would incur additional I/O at query time.

E Further results on static hash sets

Figures 5–7 show extended results of the kind shown in Figure 4 including the time to quere the hashsets in a plain for loop (rather than with prefetching) as well as multi-threaded results.

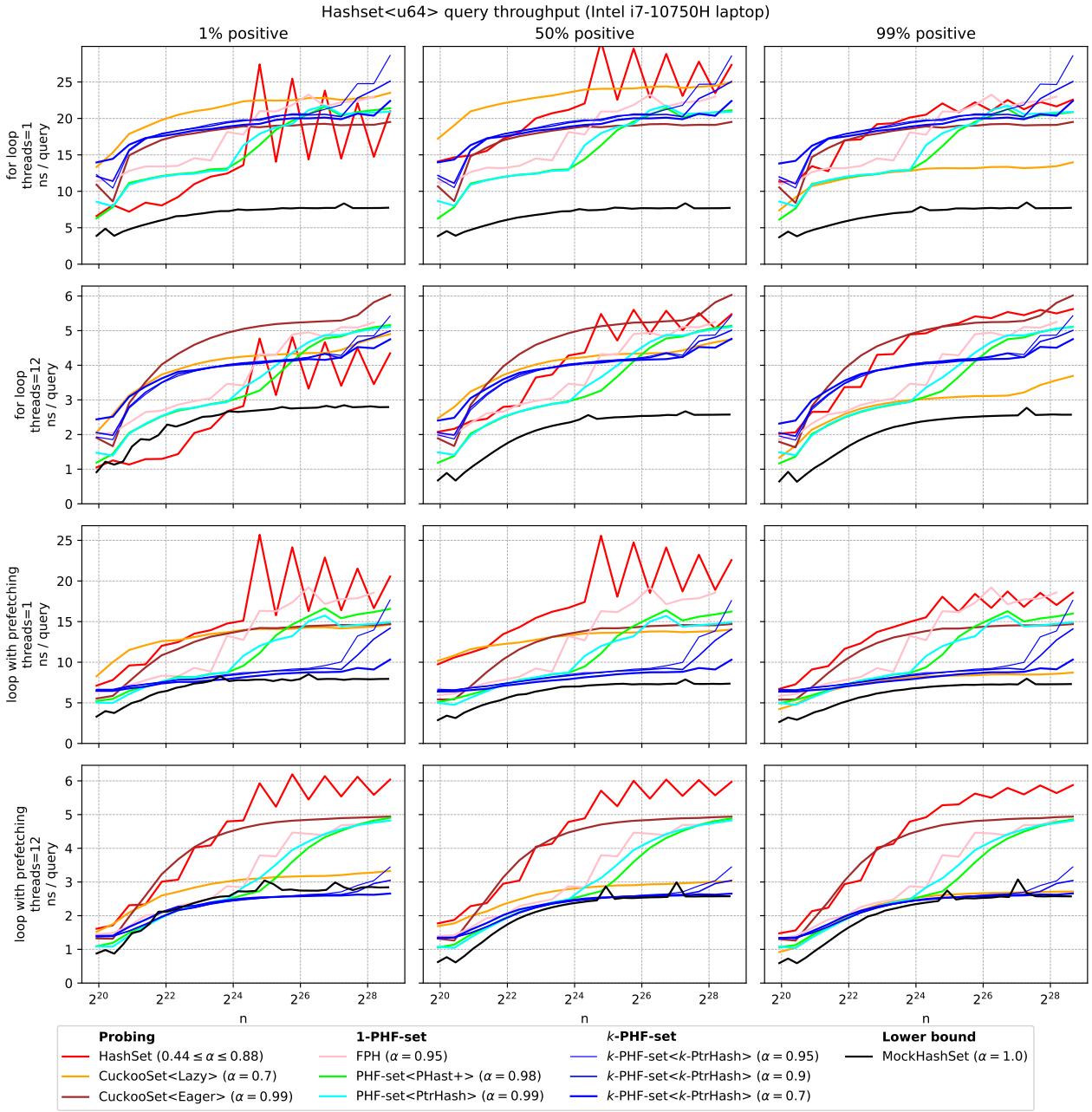

Figure 5 Results on 1 and 12 threads for the throughput of a for loop and loop with prefetching on an Intel Skylake laptop CPU.

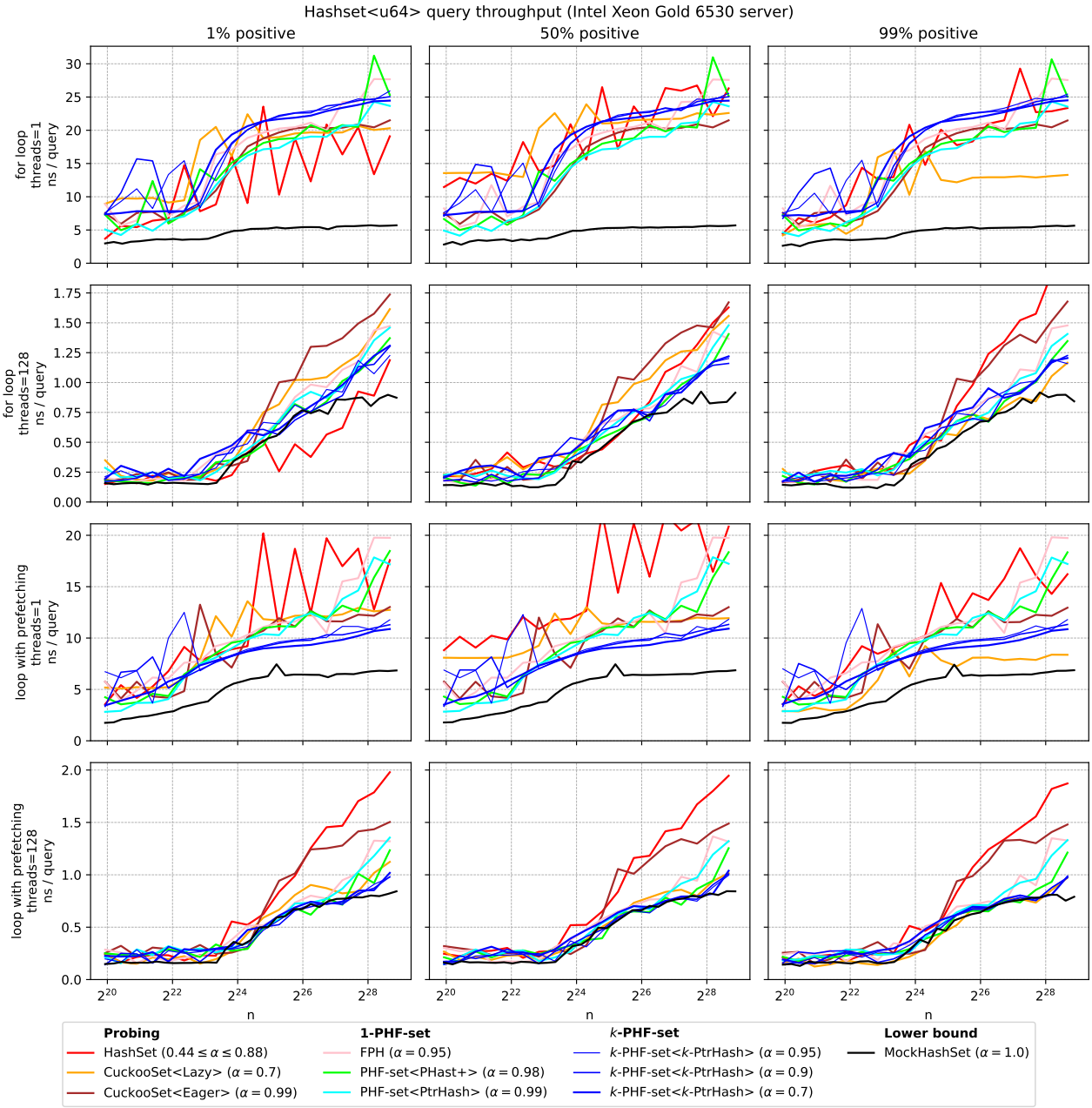

Figure 6 Results on 1 and 128 threads for the throughput of a for loop and loop with prefetching on an Intel Xeon server CPU.

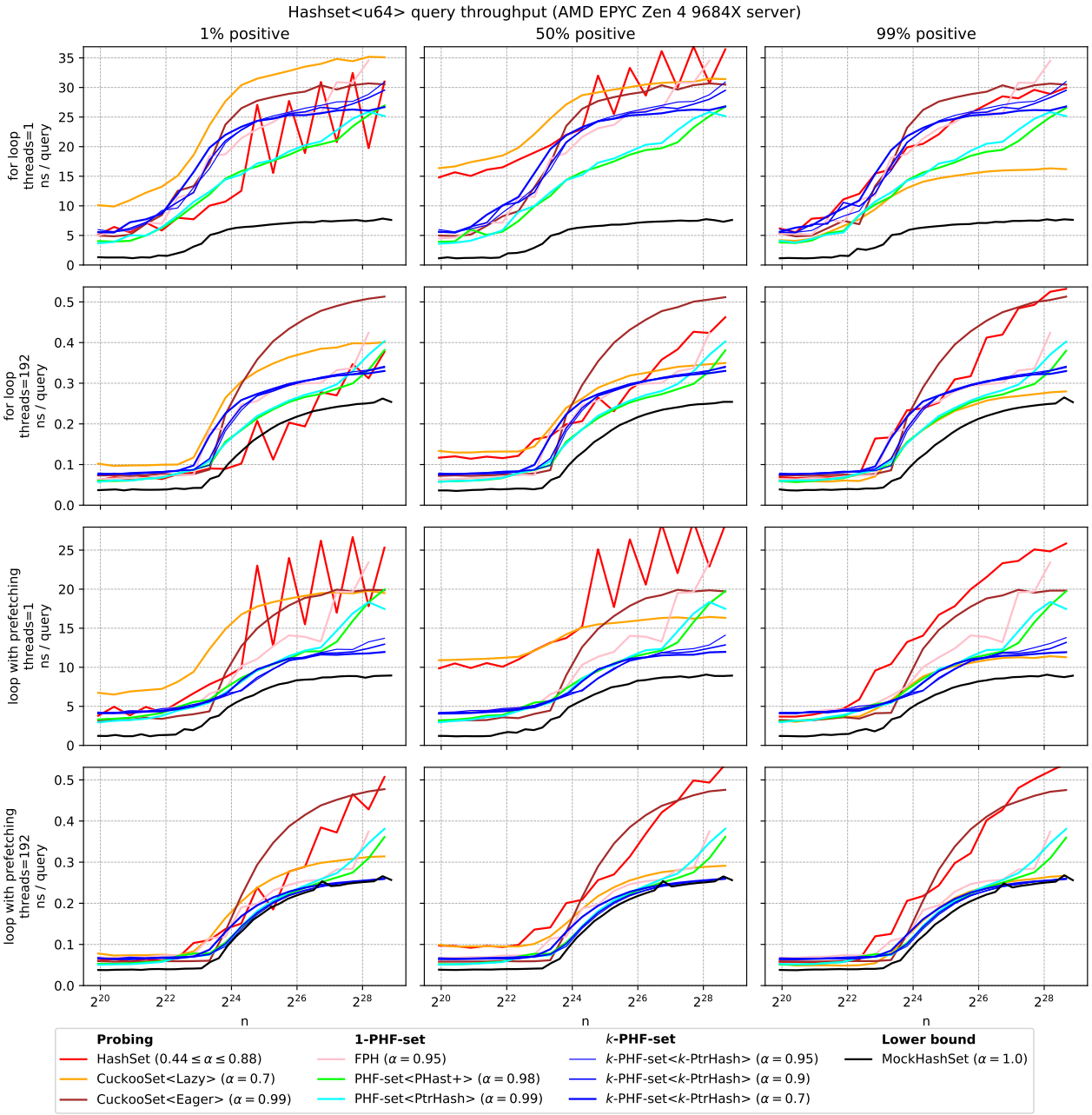

Figure 7 Results on 1 and 192 threads for the throughput of a for loop and loop with prefetching on an AMC EPYC server CPU.