Fast Succinct Retrieval and Approximate Membership using Ribbon

Peter C. Dillinger
Meta
peterd@fb.com

Lorenz Hübschle-Schneider
Karlsruhe Institute of Technology, Germany
huebschle@4z2.de

Peter Sanders
Karlsruhe Institute of Technology, Germany
sanders@kit.edu

Stefan Walzer
Cologne University
walzer@cs.uni-koeln.de

Abstract

A *retrieval* data structure for a static function $f : S \to \{0,1\}^r$ supports queries that return $f(x)$ for any $x \in S$. Retrieval data structures can be used to implement a static approximate membership query data structure (AMQ), i.e., a Bloom filter alternative, with false positive rate 2^{-r}. The information-theoretic lower bound for both tasks is $r|S|$ bits. While *succinct* theoretical constructions using $(1 + o(1))r|S|$ bits were known, these could not achieve very small overheads in practice because they have an unfavorable space–time tradeoff hidden in the asymptotic costs or because small overheads would only be reached for physically impossible input sizes. With *bumped ribbon retrieval (BuRR)*, we present the first practical succinct retrieval data structure. In an extensive experimental evaluation BuRR achieves space overheads well below 1 % while being faster than most previously used retrieval data structures (typically with space overheads at least an order of magnitude larger) and faster than classical Bloom filters (with space overhead $\geq 44\,\%$). This efficiency, including favorable constants, stems from a combination of simplicity, word parallelism, and high locality.

We additionally describe *homogeneous ribbon filter AMQs*, which are even simpler and faster at the price of slightly larger space overhead.

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

Keywords and phrases AMQ, Bloom filter, dictionary, linear algebra, randomized algorithm, retrieval data structure, static function data structure, succinct data structure, perfect hashing

Related Version There is a preprint [21] and an earlier technical report [22].

Supplement Material The code and scripts used for our experiments are available under a permissive license at github.com/lorenzhs/BuRR and github.com/lorenzhs/fastfilter_cpp.

Funding *Stefan Walzer*: DFG grant WA 5025/1-1.

1 Introduction

A *retrieval data structure* (sometimes called "static function") represents a function $f : S \to \{0,1\}^r$ for a set $S \subseteq \mathcal{U}$ of n keys from a universe $\mathcal{U}$ and $r \in \mathbb{N}$. A query for $x \in S$ must return $f(x)$, but a query for $x \in \mathcal{U} \setminus S$ may return any value from $\{0,1\}^r$.

The information-theoretic lower bound for the space needed by such a data structure is nr bits in the general case.[1] This significantly undercuts the $\Omega((\log|\mathcal{U}| + r)n)$ bits[2] needed by a dictionary, which must return "None" for $x \in \mathcal{U} \setminus S$. The intuition is that dictionaries have to store $f \subseteq \mathcal{U} \times \{0,1\}^r$ as a set of key-value pairs while retrieval data structures, surprisingly, need not store the keys. We say a retrieval data structure using s bits has *(space) overhead* $\frac{s}{nr} - 1$.

The starting point for our contribution is a *compact* retrieval data structure from [20], i.e. one with overhead $\mathcal{O}(1)$. After minor improvements, we first obtain *standard ribbon retrieval*. All theoretical analysis assumes computation on a word RAM with word size $\Omega(\log n)$ and that hash functions behave like random functions.[3] The *ribbon width* w is a parameter that also plays a role in following variants.

▶ **Theorem 1** (similar to [20]). *For any $\varepsilon > 0$, an r-bit standard ribbon retrieval data structure with ribbon width $w = \frac{\log n}{\varepsilon}$ has construction time $\mathcal{O}(n/\varepsilon^2)$, query time $\mathcal{O}(r/\varepsilon)$ and overhead $\mathcal{O}(\varepsilon)$.*

We then combine standard ribbon retrieval with the idea of *bumping*, i.e., a convenient subset $S' \subseteq S$ of keys is handled in the first *layer* of the data structure and the small rest is *bumped* to recursively constructed subsequent layers. The resulting *bumped ribbon retrieval* (BuRR) data structure has much smaller overhead for any given ribbon width w.

▶ **Theorem 2.** *An r-bit BuRR data structure with ribbon width $w = \mathcal{O}(\log n)$ and $r = \mathcal{O}(w)$ has expected construction time $\mathcal{O}(nw)$, space overhead $\mathcal{O}(\frac{\log w}{rw^2})$, and query time $\mathcal{O}(1 + \frac{rw}{\log n})$.*

In particular, BuRR can be configured to be *succinct*, i.e., can be configured to have an overhead of $o(1)$ while retaining constant access time for small r. Construction time is slightly superlinear. Note that succinct retrieval data structures were known before, even with asymptotically optimal construction and query times of $\mathcal{O}(n)$ and $\mathcal{O}(1)$, respectively [40, 4]. Seeing the advantages of BuRR requires a closer look. Details are given in Section 5, but the gist can be seen from Table 1: Among the previous succinct retrieval data structures (overheads set in bold font), only [18] can achieve small overhead in a *tunable* way, i.e., independently of n using an appropriate tuning parameter $C = \omega(\log n)$. However, this approach suffers from comparatively high constructions times. [40] and [4] are not tunable and only *barely* succinct with significant overhead in practice. A quick calculation to illustrate: Neglecting the factors hidden by $\mathcal{O}$-notation, the overheads are $\frac{\log\log n}{\sqrt{\log n}}$ and $\frac{\log^2 \log n}{r \log n}$, which is at least 75% and 7% for $r = 8$ and any $n \leq 2^{64}$. A similar estimation for BuRR with $w = \Theta(\log n)$ suggests an overhead of $\frac{\log\log n}{r \log^2 n} \approx 0.1\%$ already for $r = 8$ and $n = 2^{24}$. Moreover, by tuning the ribbon width w, a wide range of trade-offs between small overhead and fast running times can be achieved.

Overall, we believe that asymptotic analyses struggle to tell the full story due to the extremely slow decay of some "$o(1)$" terms. We therefore accompany the theoretical account

[1] If f has low entropy then *compressed static functions* [31, 4, 28] can do better and even machine learning techniques might help; see e.g. [42].

[2] This lower bound holds when $|\mathcal{U}| = \Omega(n^{1+\delta})$ for $\delta > 0$. The general bound is $\log\binom{|\mathcal{U}|}{n} + nr$ bits.

[3] This is a standard assumption in many papers and can also be justified by standard constructions [17].

		Year	$t_{\text{construct}}$	t_{query}	multiplicative overhead	shard size	Solver
Standard Ribbon	[34]	2001	$\mathcal{O}(n \log k)$	$\mathcal{O}(\log k)^\dagger$	$\frac{1}{k}$	–	peeling
	[40]	2009	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\boldsymbol{\mathcal{O}(\frac{\log\log n}{(\log n)^{1/2}})}$	$\sqrt{\log n}$	lookup table
	[9]	2013	$\mathcal{O}(n)$	$\mathcal{O}(1)$	0.2218	–	peeling
	[4]	2013	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\boldsymbol{\mathcal{O}(\frac{\log^2 \log n}{r \log n})}$	$\mathcal{O}(\frac{\log^2 \log n}{r \log n})$	–
	[39]	2014	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\Omega(1/r)$	$\mathcal{O}(1)$	sorting/sharding
	[27]	2016	$\mathcal{O}(nC^2)$	$\mathcal{O}(1)$	$0.024 + \mathcal{O}(\frac{\log n}{C})$	C	structured Gauss
⟶	[20]	2019	$\mathcal{O}(n/\varepsilon^2)$	$\mathcal{O}(r/\varepsilon)$	ε	–	Gauss
	[20]	2019	$\mathcal{O}(n/\varepsilon)$	$\mathcal{O}(r)$	$\varepsilon + \mathcal{O}(\frac{\log n}{n^\varepsilon})$	n^ε	Gauss
	[18]	2019	$\mathcal{O}(nC^2)$	$\mathcal{O}(r)$	$\boldsymbol{\mathcal{O}(\frac{\log n}{C})}$	C	structured Gauss
	[44]	2021	$\mathcal{O}(nk)$	$\mathcal{O}(k)$	$(1 + o_k(1))\mathrm{e}^{-k}$	–	peeling
BuRR			$\mathcal{O}(nw)$	$\mathcal{O}(1 + \frac{rw}{\log n})$	$\boldsymbol{\mathcal{O}(\frac{\log w}{rw^2})}$	–	on-the-fly Gauss
↪ with $w = \Theta(\log n)$:			$\mathcal{O}(n \log n)$	$\mathcal{O}(r)$	$\boldsymbol{\mathcal{O}(\frac{\log\log n}{r \log^2 n})}$	–	on-the-fly Gauss

† Expected query time. Worst case query time is $\mathcal{O}(D)$.

Table 1 Performance of various r-bit retrieval data structures with $r = \mathcal{O}(\log n)$. **Bold** overhead indicates that the data structure is (or can be configured to be) succinct. The parameters $k \in \mathbb{N}$ and $\varepsilon > 0$ are constants with respect to n. The parameter $C \in \mathbb{N}$ is typically n^α for constant $\alpha \in (0, 1)$.

with experiments comparing BuRR to other efficient (compact or succinct) retrieval data structures. We do this in the use case of data structures for approximate membership and also invite competitors not based on retrieval into the ring such as (blocked) Bloom filters and Cuckoo filters.

Data structures for approximate membership. Retrieval data structures are an important basic tool for building compressed data structures. Perhaps the most widely used application is associating an r-bit fingerprint with each key from a set $S \subseteq \mathcal{U}$, which allows implementing an *approximate membership query data structure* (*AMQ*, aka *Bloom filter replacement* or simply *filter*) that supports membership queries for S with *false positive rate* $\varphi = 2^{-r}$. A membership query for a key $x \in \mathcal{U}$ will simply compare the fingerprint of x with the result returned by the retrieval data structure for x. The values will be the same if $x \in S$. Otherwise, they are the same only with probability 2^{-r}.

In addition to the AMQs following from standard ribbon retrieval and BuRR, we also present homogeneous ribbon filters, which are not directly based on retrieval.

▶ **Theorem 3.** *Let $r \in \mathbb{N}$ and $\varepsilon \in (0, \frac{1}{2}]$. There is $w \in \mathbb{N}$ with $\frac{w}{\max(r, \log w)} = \mathcal{O}(1/\varepsilon)$ such that the homogeneous ribbon filter with ribbon width w has false positive rate $\varphi \leq (1 + \varepsilon^2)2^{-r}$ and space overhead $\mathcal{O}(\varepsilon)$. On a word RAM with word size $\geq w$ expected construction time is $\mathcal{O}(n/\varepsilon)$ and query time is $\mathcal{O}(r)$.*

Experiments. Figure 1 shows some of the results explained in detail later in the paper. In the depicted parallel setting, ribbon-based AMQs (blue) are the fastest static AMQs when an overhead less than $\approx 44\%$ is desired (where "fastest" considers a somewhat arbitrary weighting of construction and query times). The advantage is less pronounced in the sequential setting.

Why care about space? Especially in AMQ applications, retrieval data structures occupy a considerable fraction of RAM in large server farms continuously drawing many megawatts of power. Even small reductions (say 10 %) in their space consumption thus translate into considerable cost savings. Whether or not these space savings should be pursued at the price of increased access costs depends on the number of queries per second. The lower the

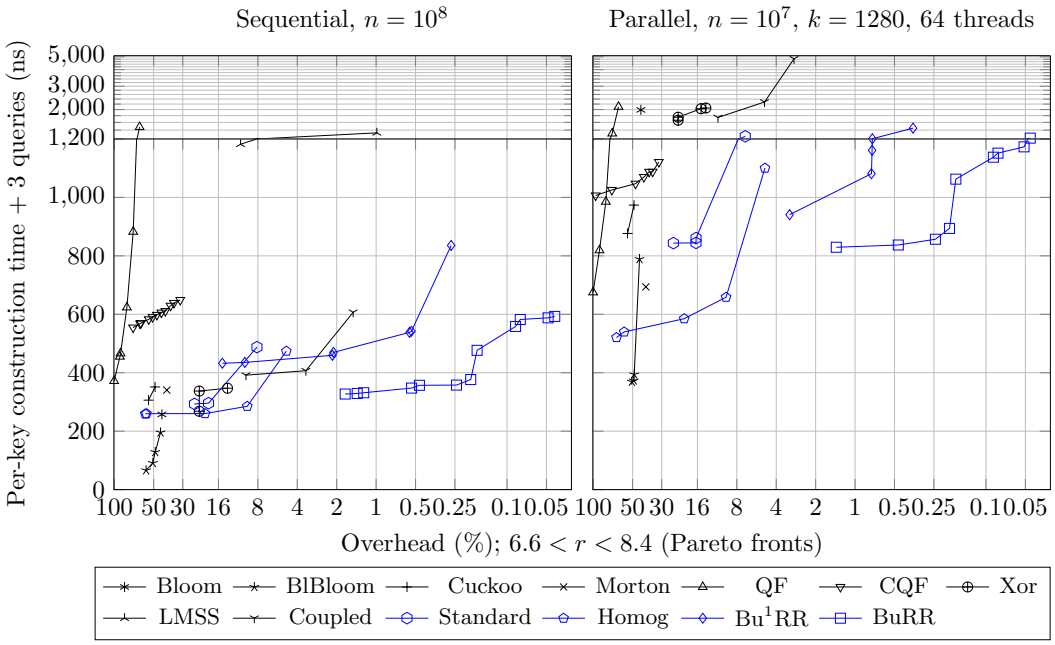

Figure 1 Performance–overhead trade-off for measured false-positive rate in 0.003–0.01 (i.e., $r \approx 8$), for different AMQs and inputs. Ribbon-based data structures are in blue. For each category of approaches, only variants are shown that are not Pareto-dominated by variants in the same category. Sequential benchmarks use a single filter of size n while the parallel benchmark uses 1280 filters of size n and utilizes 64 cores. Logarithmic vertical axis above 1200 ns.

access frequency, the more worthwhile it is to occasionally spend increased access costs for a permanently lowered memory budget. Since the false-positive rate also has an associated cost (e.g. additional accesses to disk or flash) it is also subject to tuning. The entire set of Pareto-optimal variants with respect to tradeoffs between space, access time, and FP rate is relevant for applications. For instance, sophisticated implementations of LSM-trees use multiple variants of AMQs at once based on known access frequencies [14]. Similar ideas have been used in compressed data bases [38].

Outline. The paper is organized as follows (section numbers in parentheses). After important preliminaries (2), we explain our data structures and algorithms in broad strokes (3) and summarize our experimental findings (4). We then summarize related work (5). In the full paper [21] we give a detailed theoretical analysis, extensively describe the design space of BuRR, and discuss additional experiments.

2 Linear Algebra Based Retrieval Data Structures and SGAUSS

A simple, elegant and highly successful approach for compact and succinct retrieval uses linear algebra over the finite field $\mathbb{Z}_2 = \{0, 1\}$ [16, 27, 1, 40, 12, 9, 18, 20]. Refer to Section 5 for a discussion of alternative and complementary techniques.

The train of thought is this: A natural idea would be to have a hash function point to a location where the key's information is stored while the key itself need not be stored. This fails because of hash collisions. We therefore allow the information for each key to be dispersed over several locations. Formally we store a table $Z \in \{0, 1\}^{m \times r}$ with $m \geq n$ entries of r bits each and to define $f(x)$ as the bit-wise xor of a set of table entries whose positions

$h(x) \subseteq [m]$ are determined by a hash function h.[4] This can be viewed as the matrix product $\vec{h}(x)Z$ where $\vec{h}(x) \in \{0,1\}^m$ is the characteristic (row)-vector of $h(x)$. For given h, the main task in building the data structure is to find the right table entries such that $\vec{h}(x)Z = f(x)$ holds for every key x. This is equivalent to solving a system of linear equations $AZ = \mathbf{b}$ where $A = (\vec{h}(x))_{x \in S} \in \{0,1\}^{n \times m}$ and $\mathbf{b} = (f(x))_{x \in S} \in \{0,1\}^{n \times r}$. Note that rows in the constraint matrix A correspond to keys in the input set S. In the following, we will thus switch between the terms "row" and "key" depending on which one is more natural in the given context.

An encouraging observation is that even for $m = n$, the system $AZ = \mathbf{b}$ is solvable with constant probability if the rows of A are chosen uniformly at random [13, 40]. With linear query time and cubic construction time, we can thus achieve optimal space consumption. For a practically useful approach, however, we want the 1-entries in $\vec{h}(x)$ to be sparse and highly localized to allow cache-efficient queries in (near) constant time and we want a (near) linear time algorithm for solving $AZ = \mathbf{b}$. This is possible if $m > n$.

A particularly promising approach in this regard is SGAUSS from [20] that chooses the 1-entries within a narrow range. Specifically, it chooses w random bits $c(x) \in \{0,1\}^w$ and a random *starting position* $s(x) \in [m - w - 1]$, i.e., $\vec{h}(x) = 0^{s(x)-1}c(x)0^{m-s(x)-w+1}$. For $m = (1+\varepsilon)n$ some value $w = \mathcal{O}(\log(n)/\varepsilon)$ suffices to make the system $AZ = \mathbf{b}$ solvable with high probability. We call w the *ribbon width* because after sorting the rows of A by $s(x)$ we obtain a matrix which is not technically a band matrix, but which likely has all 1-entries within a narrow *ribbon* close to the diagonal. The solution Z can then be found in time $\mathcal{O}(n/\varepsilon^2)$ using Gaussian elimination [20] and bit-parallel row operations; see also Figure 2 (a).

3 Ribbon Retrieval and Ribbon Filters

We advance the linear algebra approach to the point where space overhead is almost eliminated while keeping or improving the running times of previous constructions.

Ribbon solving. Our first contribution is a simple algorithm we could not resist to also call *ribbon* as in ***R**apid **I**ncremental **B**oolean **B**anding **ON** the fly*. It maintains a system of linear equations in row echelon form as shown in Figure 2 **(b)**. It does so *on-the-fly*, i.e. while equations arrive one by one in arbitrary order. For each index i of a column there may be at most one equation that has its leftmost one in column i. When an equation with row vector a arrives and its slot is already taken by a row a', then ribbon performs the row operation $a \leftarrow a \oplus a'$, which eliminates the 1 in position i, and continues with the modified row. An invariant is that rows have all their nonzeroes in a range of size w, which allows to process rows with a small number of bit-parallel word operations. This insertion process is *incremental* in that insertions do not modify existing rows. This improves performance and allows to cheaply roll back the most recent insertions which will be exploited below. It is a non-trivial insight that the order in which equations are added does not significantly affect the expected number of row operations. This is made precise and proved in the full paper.

When all rows are processed we perform back-substitution to compute the solution matrix Z. At least for small r, *interleaved representation* of Z works well, where blocks of size $w \times r$ of Z are stored column-wise. A query for x can then retrieve one bit of $f(x)$ at a time by applying a population count instruction to pieces of rows retrieved from at most two of

[4] In this paper, $[k]$ can stand for $\{0, \ldots, k-1\}$ or $\{1, \ldots, k\}$ (depending on the context), and $a..b$ stands for $\{a, \ldots, b\}$.

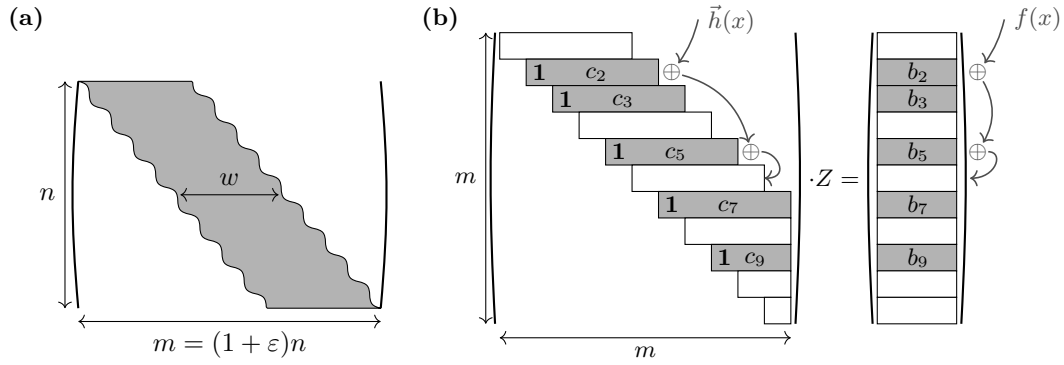

Figure 2 **(a)** Typical shape of the random matrix A with rows $(\vec{h}(x))_{x\in S}$ sorted by starting positions. The shaded "ribbon" region contains random bits. Gaussian elimination never causes any fill-in outside of the ribbon.
(b) Shape of the linear system M in row echelon form maintained using Boolean banding on the fly. In gray we visualize the insertion of a key x where (i) $\vec{h}(x)$ has its left-most 1 in position $s(x) = 2$, (ii) after xoring the second row of M to $\vec{h}(x)$, the left-most 1 is in position 5 and (iii) xoring the fifth row as well, the left-most 1 is in position 6. The resulting row fills the previously empty sixth row of M and $f(x) \oplus b_2 \oplus b_5$ is added as right hand side.

these blocks. This is particularly advantageous for negative queries to AMQs (i.e. queries of elements not in the set), where only two of r bits need to be retrieved on average. More details are given in the full paper.

3.1 Standard Ribbon

When employing no further tricks, we obtain *standard ribbon retrieval*, which is essentially the same data structure as in [20] except with a different solver that is faster by noticeable constant factors. A problem is that w becomes prohibitively large when n is large and ε is small. For example, experiments show that for $\varepsilon \leq 3.5\%$ and construction success rate $\geq 50\%$, standard word size $w = 64$ only scales to around $n \leq 10^4$ and more expensive $w = 128$ only scales to around $n \leq 10^6$. To some degree this can be mitigated by sharding techniques [43], but in this paper we pursue a more ambitious route.

3.2 Bumped Ribbon Retrieval

Our main contribution is *bumped ribbon retrieval (BuRR)*, which reduces the required ribbon width to a constant that only depends on the targeted space efficiency. BuRR is based on two ideas.

Bumping. The ribbon solving approach manages to insert most rows (representing most keys of S) even when w is small. Thus, by eliminating those rows/keys that cause a linear dependency, we obtain a compact retrieval data structure for a large subset of S. The remaining keys are *bumped*, meaning they are handled by a fallback data structure which, by recursion, can be a BuRR data structure again. We show that only $\mathcal{O}(\frac{n \log w}{w})$ keys need to be bumped in expectation. Thus, after a constant number of layers (we use 4), a less ambitious retrieval data structure can be used to handle the few remaining keys without bumping.

The main challenge is that we need additional metadata to encode which keys are bumped. The basic *bumped retrieval* approach is adopted from the updateable retrieval data structure

filtered retrieval (FiRe) [39]. To shrink the input size by a moderate constant factor, FiRe needs a constant number of bits per key (around 4). This leads to very high space overhead for small r. A crucial observation for BuRR is that bumping can be done with granularity much coarser than per-key. We will bump keys based on their starting position and say *position i is bumped* to indicate that all keys with $s(x) = i$ are bumped. Bumping by position is sufficient because linear dependencies in A are largely unrelated to the actual bit patterns $c(x)$ but mostly caused by fluctuations in the number of keys mapped to different parts of the matrix A. By selectively bumping ranges of positions in overloaded parts of the system, we can obtain a solvable system. Furthermore, our analysis shows that we can drastically limit the spectrum of possible bumping ranges; see below.

Overloading. Besides metadata, space overhead results from the $m - n + n_b$ excess slots of the table where n_b is the number of bumped keys. Trying out possible values of $\varepsilon = \frac{m-n}{n} > 0$ one sees that the overhead due to excess slots is always $\Omega(1/w)$ and will thus dominate the overhead due to metadata. However, we show that by choosing $\varepsilon < 0$ (of order $-\varepsilon = \mathcal{O}(\frac{\log w}{w})$), i.e., by *overloading* the table, we can almost completely eliminate excess table slots so that the minuscule amount of metadata becomes the dominant remaining overhead. There are many ways to decide and encode which keys are bumped. Here, we outline a simple variant that achieves very good performance in practice and is a generalization of the theoretically analyzed approach. We expand on the much larger design space of BuRR in the full paper.

Deciding what to bump. We subdivide the possible starting positions into *buckets* of width $b = \mathcal{O}(w^2/\log w)$ and allow to bump a single initial range of each bucket. The keys (or more precisely pairs of hashes and the value to be retrieved) are sorted according to the bucket addressed by the starting position $s(x)$. We use a fast in-place integer sorter for this purpose [2]. Then buckets are processed one after the other from *left to right*. Within a bucket, however, keys are inserted into the row echelon form from *right to left*. The reason for this is that insertions of the previous bucket may have "spilled over" causing additional load on the left of the bucket – an issue we wish to confront as late as possible. See also Figure 3.

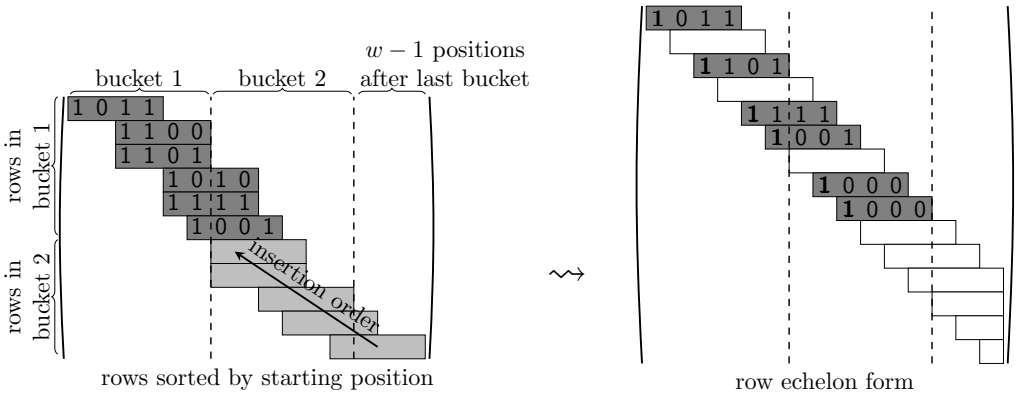

Figure 3 Illustration of BuRR construction with $n = 11$ keys, $m = 2b + w - 1 = 15$ table positions, ribbon width $w = 4$ and bucket size $b = 6$. Keys of the first bucket were successfully inserted (from right to left) into row echelon form with two insertions "overflowing" into the second bucket. Insertions of the second bucket's rows will be attempted next, in the indicated order.

If all keys of a bucket can be successfully inserted, no keys of the bucket are bumped. Otherwise, suppose the first failed insertion for a bucket $[i, i + b)$ concerns a key where $s(x) = i + k$ is the k-th position of the bucket. We *could* decide to bump all keys x' of the

bucket with $s(x') \leq i + k$, which would require storing the *threshold* k using $\mathcal{O}(\log w)$ bits and which would yield an overhead of $\mathcal{O}(\log^2(w)/w^2)$ due to metadata. *Instead*, to reduce this overhead to $\mathcal{O}(\log(w)/w^2)$, we only allow a constant number of threshold values. This means that we find the smallest threshold value t with $t \geq k$ representable by metadata and bump all keys x' with $s(x') \leq i + t$. This requires rolling back the insertions of keys x' with $s(x') \in [k, t]$ by clearing the most recently populated rows from the row echelon form. One good compromise between space and speed stores 2 bits per bucket encoding the threshold values $\{0, \ell, u, b\}$, for suitable ℓ and u. The special case $\ell = u = \frac{3}{8}w$ is used in our analysis. Another slightly more compact variant "1^+-bit" stores one bit encoding threshold values from the set $\{0, t\}$, for a suitable t, and additionally stores a hash table of exceptions for thresholds $> t$.

Running times. With these ingredients we obtain Theorem 2 stated on page 2. It implies constant query time[5] if $rw = \mathcal{O}(\log n)$ and linear construction time if $w \in \mathcal{O}(1)$. For wider ribbons, construction time is slightly superlinear. However, in practice this does not necessarily mean that BuRR is slower than other approaches with asymptotically better bounds as the factor w involves operations with very high locality. An analysis in the external memory model reveals that BuRR construction is possible with a single scan of the input and integer sorting of n objects of size $\mathcal{O}(\log n)$ bits; see the full paper for details.

3.3 Homogeneous Ribbon Filter

For the application of ribbon to AMQs, we can also compute a *uniformly random* solution of the *homogeneous* equation system $AZ = 0$, i.e., we compute a retrieval data structure that will retrieve 0^r for all keys of S but is unlikely to produce 0^r for other inputs. Since $AZ = 0$ is always solvable, there is no need for bumping. The crux is that the false positive rate is no longer 2^{-r} but higher. In the full paper we show that with table size $m = (1 + \varepsilon)n$ and $\varepsilon = \Omega(\frac{\max(r, \log w)}{w})$ the difference is negligible, thereby showing Theorem 3. Homogeneous ribbon AMQs are simpler and faster than BuRR but have higher space overhead. Our experiments indicate that together, BuRR and homogeneous ribbon AMQs cover a large part of the best tradeoffs for static AMQs.

3.4 Analysis outline

To get an intuition for the relevant linear systems, it is useful to consider two simplifications. First, assume that $\vec{h}(x)$ contains a block of w uniformly random *real* numbers from $[0, 1]$ rather than w random bits. Secondly, assume that we sort the rows by starting position and use Gaussian elimination rather than ribbon to produce a row echelon form. In Figure 4 **(a)** we illustrate for such a matrix with $\times$-marks where the pivots would be placed and in yellow the entries that are eliminated (with one row operation each); both with probability 1, i.e. barring coincidences where a row operation eliminates more than one entry. The $\times$-marks trace a diagonal through the matrix except that the green column and the red row are skipped because the end of the (gray) area of nonzeroes is reached. "Column failures" correspond to free variables and therefore unused space. "Row failures" correspond to linearly dependent equations and therefore failed insertions. This view remains largely intact when handling *Boolean* equations in *arbitrary* order except that the *ribbon diagonal*, which we introduce as

[5] It should be noted that the proof invokes a lookup table in one case to speed up the computation of a matrix vector product. In Section 5, we argue that lookup tables should be avoided in practice. Technically, our *implementation* using *interleaved representation* has a query time of $\mathcal{O}(r)$.

an analogue to the trace of pivot positions, has a more abstract meaning and probabilistically suffers from row and column failures depending on its *distance* to the ribbon border.

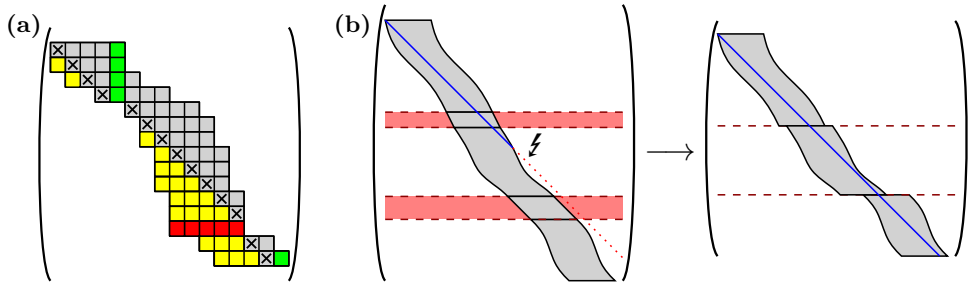

Figure 4 **(a)** The simplified ribbon diagonal (made up of ×-marks) passing through A. **(b)** The idea of BuRR: When starting with an "overloaded" linear system and removing sets of rows strategically, we can often ensure that the ribbon diagonal does not collide with the ribbon border (except possibly in the beginning and the end).

The idea of standard ribbon is to give the gray ribbon area an expected slope of less than 1 such that row failures are unlikely. BuRR, as illustrated in Figure 4 **(b)** largely avoids both failure types by using a slope bigger than 1 but removing ranges of rows in strategic positions. Homogeneous ribbon filters, despite being the simplest approach, have the most subtle analysis as both failure types are allowed to occur. While row failures cannot cause outright construction failure, they are linked to a compromised false positive rate in a non-trivial way. Our proofs involve mostly simple techniques as would be used in the analysis of linear probing, which is unsurprising given that [20] has already established a connection to Robin Hood hashing. We also profit from queuing theory via results we import from [20].

3.5 Further results

We have several further results around variants of BuRR that we summarize here. See the full paper for detail.

Perhaps most interesting is **bump-once ribbon retrieval (Bu^{1}RR)**, which improves the worst-case query time by guaranteeing that each key can be retrieved from one out of two layers – its *primary layer* or the next one. The primary layer of the keys is now distributed over all layers (except for the last). When building a layer, the keys bumped from the previous layer are inserted into the row echelon form first. The layer sizes have to be chosen in such a way that no bumping is needed for these keys with high probability. Only then are the keys with the current layer as their primary layer inserted – now allowing bumping.

For building large retrieval data structures, **parallel construction** is important. Doing this directly is difficult for ribbon retrieval since there is no efficient way to parallelize back-substitutions. However, we can partition the equation system into parts that can be solved independently by bumping w consecutive positions. Note that this can be done transparently to the query algorithm by using the bumping mechanism that is present anyway.

For large r, we accelerate queries by working with **sparse bit patterns** that set only a small fraction of the w bits in the window used for BuRR. In some sense, we are covering here the middle ground between ribbon and spatial coupling [44]. Experiments indicate that setting 8 out of 64 bits indeed speeds up queries for $r \in \{8, 16\}$ at the price of increased (but still small) overhead. Analysis and further exploration of this middle ground may be an interesting area for future work.

4 Summary of Experimental Findings

We performed extensive experiments to evaluate our ribbon-based data structures and competitors. We summarize our findings here with details provided in the full paper.

Implementation Details. We implemented BuRR in C++ using template parameters that allow us to navigate a large part of the design space mapped in the full paper. (Recall that r is the retrieval width, ϵ the overloading factor, w the ribbon width, and b the bucket width; t, ℓ, and u are bumping thresholds.) Input keys themselves are only hashed once to a 64-bit *master-hash-code (MHC)* that is subsequently used when further hash values are needed. For this, fast linear congruential mapping is used. The table is stored in an *interleaved fashion*, i.e., it is organized as rm/w words of w bits each where word i represents bit $i \bmod r$ of w subsequent table entries. This organization allows the extraction of one retrieved bit from two adjoining machine words using population-count instructions. Interleaved representation is advantageous for uses of BuRR as an AMQ data structure since a negative query only has to extract two bits in expectation. Moreover, the implementation directly works for any value of r. The default data structure has four layers, the last of which uses $w' := \min(w, 64)$ and $\varepsilon \geq 0$, where ε is increased in increments of 0.05 until no keys are bumped. For 1^+-bit, we choose $t := \lceil -2\varepsilon b + \sqrt{b/(1+\varepsilon)}/2 \rceil$ and $\varepsilon := -2/3 \cdot w/(4b+w)$. For 2-bit, parameter tuning showed that $\ell := \lceil (0.13 - \varepsilon/2)b \rceil$, $u := \lceil (0.3 - \varepsilon/2)b \rceil$, and $\varepsilon := -3/w$ work well for $w = 32$; for $w \geq 64$, we use $\ell = \lceil (0.09 - 3\varepsilon/4)b \rceil$, $u = \lceil (0.22 - 1.3\varepsilon)b \rceil$, and $\varepsilon := -4/w$.

In addition, there is a prototypical implementation of Bu[1]RR from [22]. Both BuRR and Bu[1]RR build on the same software for ribbon solving from [22]. For validation we extend the experimental setup used for Cuckoo and Xor filters [29], with our code and scripts available at github.com/lorenzhs/fastfilter_cpp and github.com/lorenzhs/BuRR.

Experimental Setup. All experiments were run on a machine with an AMD EPYC 7702 processor with 64 cores, a base clock speed of 2.0 GHz, and a maximum boost clock speed of 3.35GHz. The machine is equipped with 1 TiB of DDR4-3200 main memory and runs Ubuntu 20.04. We use `clang++` 11.0 with optimization flags `-O3 -march=native`. During sequential experiments, only a single core was used at any time to minimize interference.

We looked at different input sizes $n \in \{10^6, 10^7, 10^8\}$. Like most studies in this area, we first look at a **sequential** workload on a powerful processor with a considerable number of cores. However, this seems unrealistic since in most applications, one would not let most cores lay bare but use them. Unless these cores have a workload with very high locality this would have a considerable effect on the performance of the AMQs. We therefore also look at a scenario that might be the opposite extreme to a sequential unloaded setting. We run the benchmarks on all available hardware threads in **parallel**. Construction builds many AMQs of size n in parallel. Queries select AMQs randomly. This emulates a large AMQ that is parallelized using sharding and puts very high load on the memory system.

Experimental Results. Two preliminary remarks are in order: Firstly, since every retrieval data structure can be used as a filter but not vice versa, our experiments are for filters, which admits a larger number of competitors. Secondly, to reduce complexity (for now), our speed ranking considers the sum of construction time per key and three query times.[6]

Space Overhead of BuRR Figure 5 plots the fraction e of empty slots of BuRR for $w = 64$ and several combinations of bucket size b and different threshold compression

[6] Queries measured in three settings: Positive keys, negative keys and a mixed data set (50 % chance of being positive). The latter is not an average of the first two due to branch mispredictions. In the appendix, we also measure the individual operations resulting in similar conclusions.

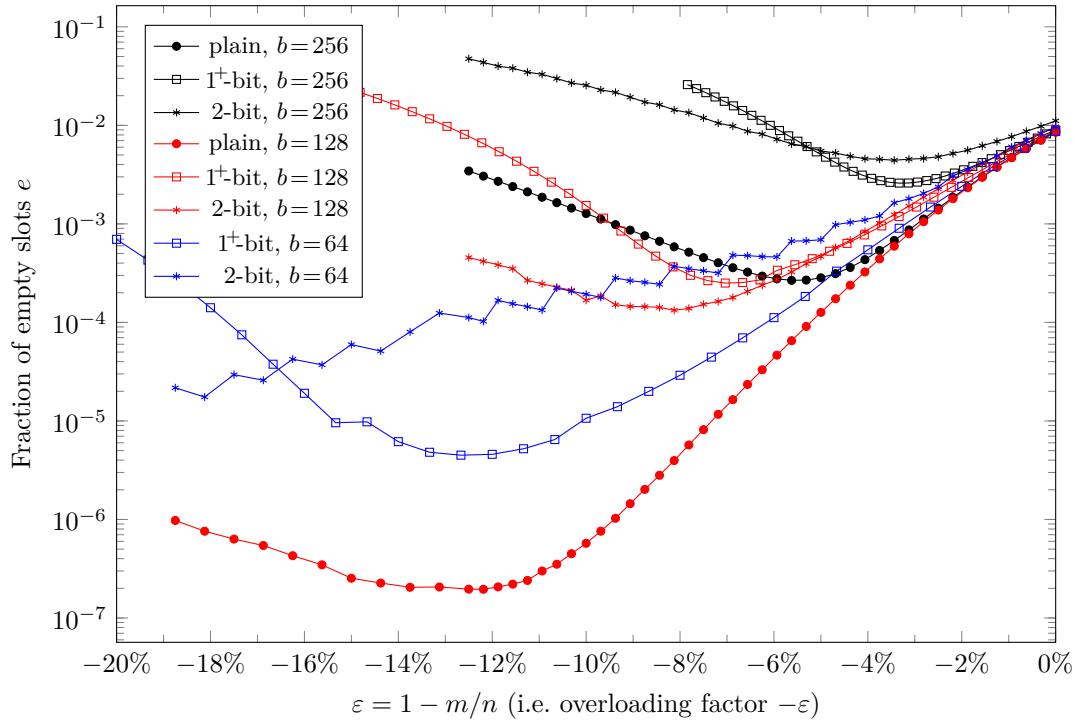

Figure 5 Fraction of empty slots for various configurations of bumped ribbon retrieval with $w = 64$, depending on the overloading factor $-\varepsilon$.

schemes. Similar plots are given in the full paper for $w = 32$, $w = 128$, and for $w = 64$ with sparse coefficients. Note that (for an infinite number of layers), the overhead is about $o = e + \mu/(rb(1-e))$ where r is the number of retrieved bits and μ is the number of metadata bits per bucket. Hence, at least when μ is constant, the overhead is a monotonic function in e and thus minimizing e also minimizes overhead.

We see that for small $|\varepsilon|$, e decreases exponentially. For sufficiently small b, e can get almost arbitrarily small. For fixed $b > w$, e eventually reaches a local minimum because with threshold-based compression, a large overload enforces large thresholds ($> w$) and thus empty regions of buckets. Which actual configuration to choose depends primarily on r. Roughly, for larger r, more and more metadata bits (i.e., small b, higher resolution of threshold values) can be invested to reduce e. For fixed b and threshold compression schemes, one can choose ε to minimize e. One can often choose a larger ε to get slightly better performance due to less bumping with little impact on o. Perhaps the most delicate tuning parameters are the thresholds to use for 2-bit and 1^+-bit compression. Indeed, in Figure 5 1^+-bit compression has lower e than 2-bit compression for $b = 64$ but higher e for $b = 128$. We expect that 2-bit compression could always achieve smaller e than 1^+-bit compression, but we have not found choices for the threshold values that always ensure this.

Ribbon yields the fastest static AMQs for overhead $< 44\%$. Consider Figure 1 on page 4, where we show the tradeoff between space overhead and computation cost for a range of AMQs for false positive rate $\varphi \approx 2^{-8}$ (i.e., $r = 8$ for BuRR) and large inputs.[7] In the

[7] Small deviations of parameters are necessary because not all filters support arbitrary parameter choices. Also note that different filters have different functionality: (Blocked) Bloom allows dynamic insertion,

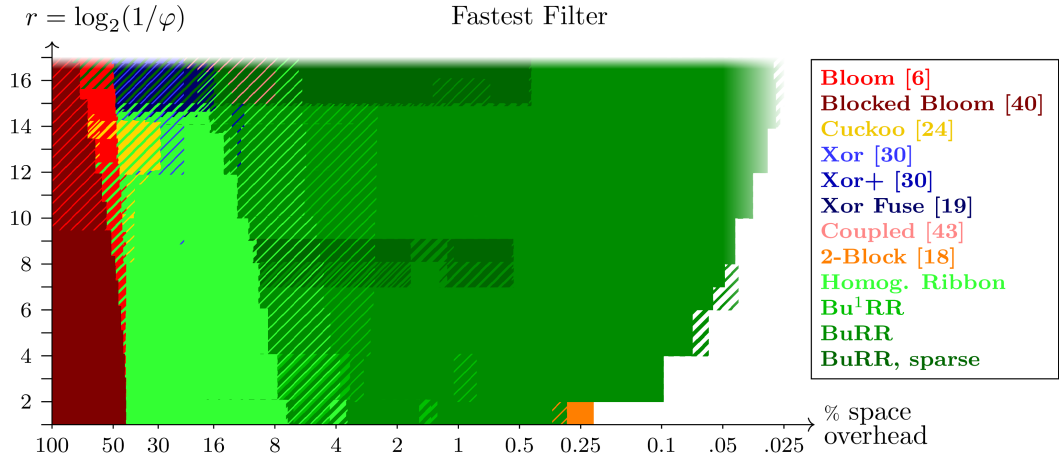

Figure 6 Fastest AMQ category for different choices of overhead and false-positive rate $\varphi = 2^{-r}$. Shaded regions indicates a dependency on the input type. Ranking metric: construction time per key plus time for three queries, of which one is positive, one negative, and one mixed (50 % chance of either).

parallel workload on the right all cores access many AMQs randomly.

Only three AMQs have Pareto-optimal configurations for this case: BuRR for space overhead below 5 % (actually achieving between 1.4 % and 0.2 % for a narrow time range of 830–890 ns), homogeneous ribbon for space overhead below 44 % (actually achieving between 20 % and 10 % for a narrow time range 580–660 ns), and *blocked Bloom filters* [41] with time around 400 ns at the price of space overhead of around 50 %. All other tried AMQs are dominated by homogeneous ribbon and BuRR. Somewhat surprisingly, this even includes plain Bloom filters [6] which are slow because they incur several cache faults for each insertion and positive query. Since plain Bloom filters are extensively used in practice (often in cases where a static interface suffices), we conclude that homogeneous ribbon and BuRR are fast enough for a wide range of applications, opening the way for substantial space savings in those settings. BuRR is at least twice as fast as all tried retrieval data structures.[8] The filter data structures that support counting and deletion (Cuckoo filters [24] and the related Morton filters [10] as well as the quotient filters QF [35] and CQF [5]) are slower than the best static AMQs.

The situation changes slightly when going to a sequential workload with large inputs as shown on the left of Figure 1. Blocked Bloom and BuRR are still the best filters for large and small overhead, respectively. But now homogeneous ribbon and (variants of) the hypergraph peeling based Xor filters [30, 19] share the middle-ground of the Pareto curve between them. Also, plain Bloom filters are almost dominated by Xor filters with half the overhead. The reason is that modern CPUs can handle several main memory accesses in parallel. This is very helpful for Bloom and Xor, whose queries do little else than computing the logical (x)or of a small number of randomly chosen memory cells. Nevertheless, the faster variants of BuRR are only moderately slower than Bloom and Xor filters while having at least an order

Cuckoo, Morton and Quotient additionally allow deletion and counting. Xor [9, 19, 30, 37], Coupled [44], LMSS [34] and all ribbon variants are static retrieval data structures.

[8] FiRe [39] is likely to be faster but has two orders of magnitude higher overhead; see the full paper for more details.

of magnitude smaller overheads.

Further Results. Other claims supported by our data are:

- **Good ribbon widths are $w = 32$ and $w = 64$.** Ribbon widths as small as $w = 16$ can achieve small overhead but at least on 64-bit processors, $w \in \{32, 64\}$ seems most sensible. The case $w = 32$ is only 15–20 % faster than $w = 64$ while the latter has about four times less overhead. Thus the case $w = 64$ seems the most favorable one. This confirms that the linear dependence of the construction time on w is to some extent hidden behind the cache faults which are similar for both values of w (this is in line with our analysis in the external memory model).
- **Bu^{1}RR is slower than BuRR** by about 20 %, which may be a reasonable price for better worst-case query time in some real-time applications.[9]
- **The 1$^+$-bit variant of BuRR is smaller but slower** than the variant with 2-bit metadata per bucket, as expected, though not by a large margin.
- **Smaller inputs and smaller r change little.** For inputs that fit into cache, the Pareto curve is still dominated by blocked Bloom, homogeneous ribbon, and BuRR, but the performance penalty for achieving low overhead increases. For $r = 1$ we have data for additional competitors. GOV [28], which relies on structured Gaussian elimination, is several times slower than BuRR and exhibits an unfavorable time–overhead tradeoff. 2-block [18] uses two small dense blocks of nonzeroes and can achieve very small overhead at the cost of prohibitively expensive construction.
- **For large r, Xor filters and Cuckoo filters come into play.** Figure 6 shows the fastest AMQ depending on overhead and false positive rate $\varphi = 2^{-r}$ up to $r = 16$. While blocked Bloom, homogeneous ribbon, and BuRR cover most of the area, they lose ground for large r because their running time depends on r. Here Xor filters and Cuckoo filters make an appearance.
- **Bloom filters and Ribbon filters are fast for *negative queries*** where, on average, only two bits need to be retrieved to prove that a key is not in the set. This improves the relative standing of plain Bloom filters on large and parallel workloads with mostly negative queries.
- **Xor filters [30] and Coupled [44] have fast queries** since they can exploit parallelism in memory accesses. They suffer, however, from slow construction on large sequential inputs due to poor locality, and exhibit poor query performance when accessed from many threads in parallel. For small n, large r, and overhead between 8 % and 20 %, Coupled becomes the fastest AMQ.

5 Related Results and Techniques

We now take the time to review some related work on retrieval including all approaches listed in Table 1.

Related Problems. An important application of retrieval besides AMQs is encoding perfect *hash functions* (PHF), i.e. an injective function $p : S \to [(1+\varepsilon)|S|]$ for given $S \subseteq \mathcal{U}$. Objectives for p are compact encoding, fast evaluation and small $\varepsilon \geq 0$. Consider a result from cuckoo hashing [25, 26, 33], namely that given four hash functions $h_1, h_2, h_3, h_4 : S \to [1.024|S|]$ there exists, with high probability, a choice function $f : S \to [4]$ such that $x \mapsto h_{f(x)}(x)$ is injective. A 2-bit retrieval data structure for f therefore gives rise to a perfect hash

[9] Part of the performance difference might be due to implementation details; see the full paper.

function [9]; see also [12]. Retrieval data structures can also be used to directly store compact names of objects, e.g., in column-oriented databases [39]. This takes more space than perfect hashing but allows to encode the ordering of the keys into the names.

In retrieval for AMQs and PHFs the stored values $f(x) \in \{0,1\}^r$ are uniformly random. However, some authors consider applications where $f(x)$ has a skewed distribution and the overhead of the retrieval data structure is measured with respect to the 0-*th order empirical entropy* of f [31, 4, 28]. Note that once we can do 1-bit retrieval with low overhead, we can use that to store data with prefix-free *variable-bit-length encoding* (e.g. Huffman or Golomb codes). We can store the k-th bit of $f(x)$ as data to be retrieved for the input tuple (x,k). This can be further improved by storing R 1-bit retrieval data structures where $R = \max_{x\in S} |f(x)|$ [31, 4, 28]. By interleaving these data structures, one can make queries almost as fast as in the case of fixed r.

More Linear Algebra based approaches. It has long been known that some matrices with random entries are likely to have full rank, even when sparse [13] and density thresholds for random k-XORSAT formulas to be solvable – either at all [23, 15] or with a linear time peeling algorithm [36, 32] – have been determined.

Building on such knowledge, a solution to the retrieval problem was identified by Botelho, Pagh and Ziviani [8, 7, 9] in the context of perfect hashing. In our terminology, their rows $\vec{h}(x)$ contain 3 random 1-entries per key which makes $AZ = \boldsymbol{b}$ solvable with peeling, provided $m > 1.22n$.

Several works develop the idea from [9]. In [27, 28] only $m > 1.089n$ is needed in principle (or $m > 1.0238n$ for $|\vec{h}(x)| = 4$) but a Gaussian solver has to be used. More recently in the *spatial coupling* approach [44] $\vec{h}(x)$ has k random 1-entries within a small window, achieving space overhead $\approx e^{-k}$ while still allowing a peeling solver. With some squinting, a class of linear erasure correcting codes from [34] can be interpreted as a retrieval data structure of a similar vein, where $|\vec{h}(x)| \in \{5,\dots,k\}$ is random with expectation $\mathcal{O}(\log k)$.

Two recent approaches also based on sparse matrix solving are [18, 20] where $\vec{h}(x)$ contains two blocks or one block of random bits. Our ribbon approach builds on the latter.

We end this section with a discussion of seemingly promising techniques and give reasons why we choose not to use them in this paper. Some more details are also discussed in the experimental section of the full paper.

Shards. A widely used technique in hashing-based data structures is to use a splitting hash function to first divide the input set into many much smaller sets (shards, buckets, chunks, bins,...) that can then be handled separately [28, 3, 18, 20, 1, 40]. This incurs only linear time overhead during preprocessing and constant time overhead during a query, and allows to limit the impact of superlinear cost of further processing to the size of the shard. Even to ribbon, this could be used in multiple ways. For example, by statically splitting the table into pieces of size n^ε for standard ribbon, one can achieve space overhead $\varepsilon + \mathcal{O}(n^{-\varepsilon})$, preprocessing time $\mathcal{O}(n/\varepsilon)$, and query time $\mathcal{O}(r)$ [20]. This is, however, underwhelming on reflection. Before arriving at the current form of BuRR, we designed several variants based on sharding but never achieved better overhead than $\Omega(1/w)$. The current overhead of $\mathcal{O}\big(\log w/w^2\big)$ comes from using the splitting technique in a "soft" way – keys are assigned to buckets for the purpose of defining bumping information but the ribbon solver may implicitly allocate them to subsequent buckets.

Table lookup. The first asymptotically efficient succinct retrieval data structure we are aware of [40] uses two levels of sharding to obtain very small shards of size $\mathcal{O}\big(\sqrt{\log n}\big)$ with small asymptotic overhead. It then uses dense random matrices per shard to obtain per-shard retrieval data structures. This can be done in constant time per shard by

tabulating the solutions of all possible matrices. This leads to a multiplicative overhead of $\mathcal{O}\big(\log\log n/\sqrt{\log n}\big)$. Belazzougui and Venturini [4] use slightly larger shards of size $\mathcal{O}((1+\log\log(n)/r)\log\log(n)/\log n)$. Using carefully designed random lookup tables they show that linear construction time, constant lookup time, and overhead $\mathcal{O}\big((\log\log n)^2/\log n\big)$ is possible. We discussed on page 2 why we suspect large overhead for [40] and [4] in practice.

In general, lookup tables are often problematic for compressed data structures in practice – they cause additional space overhead and cache faults. Even if the table is small and fits into cache, this may yield efficient benchmarks but can still cause cache faults in practical workloads where the data structure is only a small part in a large software system with a large working set.

Cascaded bumping. Hash tables consisting of multiple shrinking levels are also used in *multilevel adaptive hashing* [11] and *filter hashing* [25]. While similar to BuRR in this sense, they do not maintain bumping information. This is fine for storing key-value pairs because all levels can be searched for a requested key. But it is unclear how the idea would work in the context of retrieval, i.e. without storing keys.

6 Conclusion and Future Work

BuRR is a considerable contribution to close a gap between theory and practice of retrieval and static approximate membership data structures. From the theoretical side, BuRR is succinct while achieving constant access cost for small number of retrieved bits ($r = \mathcal{O}(\log(n)/w)$). In contrast to previous succinct constructions with better asymptotic running times, its overhead is *tunable* and already small for realistic values of n. In practice, BuRR is faster than widely used data structures with much larger overhead and reasonably simple to implement. Our results further strengthen the success of linear algebra based solutions to the problem. Our on-the-fly approach shows that Gauss-like solvers can be superior to peeling-based greedy solvers even with respect to speed.

While the wide design space of BuRR leaves room for further practical improvements, we see the main open problems for large r. In practice, peeling based solvers (e.g., [44]) might outperform BuRR if faster construction algorithms can be found – perhaps using ideas like overloading and bumping. In theory, existing succinct data structures (e.g. [40, 4]) allow constant query time but have high space overhead for realistic input sizes. Combining constant cost per element for large r with small (preferably tunable) space overhead therefore remains a theoretical promise yet to be convincingly redeemed in practice.

References

1 Martin Aumüller, Martin Dietzfelbinger, and Michael Rink. Experimental variations of a theoretically good retrieval data structure. In *Proc. 17th ESA*, pages 742–751, 2009. doi:10.1007/978-3-642-04128-0_66.

2 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *CoRR*, 2020. arXiv:2009.13569.

3 Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Proc. DCC*, pages 352–361, 2014. doi:10.1109/DCC.2014.48.

4 Djamal Belazzougui and Rossano Venturini. Compressed static functions with applications. In Sanjeev Khanna, editor, *Proc. 24th SODA*, pages 229–240. SIAM, 2013. doi:10.1137/1.9781611973105.17.

5 Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't

thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012. doi:10.14778/2350229.2350275.

6 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. doi:10.1145/362686.362692.

7 Fabiano Cupertino Botelho. *Near-Optimal Space Perfect Hashing Algorithms*. PhD thesis, Federal University of Minas Gerais, 2008. URL: http://cmph.sourceforge.net/papers/thesis.pdf.

8 Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proc. 10th WADS*, pages 139–150, 2007. doi:10.1007/978-3-540-73951-7_13.

9 Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013. doi:10.1016/j.is.2012.06.002.

10 Alex D. Breslow and Nuwan Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *VLDB J.*, 29(2-3):731–754, 2020. doi:10.1007/s00778-019-00561-0.

11 Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In David S. Johnson, editor, *Proc. 1st SODA*, pages 43–53. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320181.

12 Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proc. 15th SODA*, pages 30–39. SIAM, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982797.

13 Colin Cooper. On the rank of random matrices. *Random Structures & Algorithms*, 16(2):209–232, 2000. doi:10.1002/(SICI)1098-2418(200003)16:2<209::AID-RSA6>3.0.CO;2-1.

14 Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3064054.

15 Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Proc. 37th ICALP (1)*, pages 213–225, 2010. doi:10.1007/978-3-642-14165-2_19.

16 Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Proc. 35th ICALP (1)*, pages 385–396, 2008. doi:10.1007/978-3-540-70575-8_32.

17 Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *Proc. 36th ICALP (1)*, pages 354–365, 2009. doi:10.1007/978-3-642-02927-1_30.

18 Martin Dietzfelbinger and Stefan Walzer. Constant-time retrieval with $O(\log m)$ extra bits. In *Proc. 36th STACS*, pages 24:1–24:16, 2019. doi:10.4230/LIPIcs.STACS.2019.24.

19 Martin Dietzfelbinger and Stefan Walzer. Dense peelable random uniform hypergraphs. In *Proc. 27th ESA*, pages 38:1–38:16, 2019. doi:10.4230/LIPIcs.ESA.2019.38.

20 Martin Dietzfelbinger and Stefan Walzer. Efficient Gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *Proc. 27th ESA*, pages 39:1–39:18, 2019. doi:10.4230/LIPIcs.ESA.2019.39.

21 Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. *CoRR*, abs/2106.12270, 2021. arXiv:2109.01892.

22 Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than Bloom and Xor. *CoRR*, 2021. arXiv:2103.02515.

23 Olivier Dubois and Jacques Mandler. The 3-XORSAT threshold. In *Proc. 43rd FOCS*, pages 769–778, 2002. doi:10.1109/SFCS.2002.1182002.

24 Bin Fan, David G. Andersen, and Michael Kaminsky. Cuckoo filter: Better than Bloom. *;login:*, 38(4), 2013. URL: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/cuckoo-filter-better-bloom.

25 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/s00224-004-1195-x.

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

27 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *Proc. 15th SEA*, pages 339–352, 2016. doi:10.1007/978-3-319-38851-9_23.

28 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Information and Computation*, 2020. doi:10.1016/j.ic.2020.104517.

29 Thomas Mueller Graf and Daniel Lemire. fastfilter_cpp, 2019. URL: https://github.com/FastFilter/fastfilter_cpp.

30 Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than Bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25:1–16, 2020. doi:10.1145/3376122.

31 Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh. Storing a compressed function with constant time access. In *Proc. 17th ESA*, pages 730–741, 2009. doi:10.1007/978-3-642-04128-0_65.

32 Svante Janson and Malwina J. Luczak. A simple solution to the k-core problem. *Random Struct. Algorithms*, 30(1-2):50–62, 2007. doi:10.1002/rsa.20147.

33 Marc Lelarge. A new approach to the orientation of random hypergraphs. In *Proc. 23rd SODA*, pages 251–264. SIAM, 2012. doi:10.1137/1.9781611973099.23.

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

35 Tobias Maier, Peter Sanders, and Robert Williger. Concurrent expandable AMQs on the basis of quotient filters. In *Proc. 18th SEA*, pages 15:1–15:13, 2020. doi:10.4230/LIPIcs.SEA.2020.15.

36 Michael Molloy. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005. doi:10.1002/rsa.20061.

37 Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *ACM Journal of Experimental Algorithmics*, 27, mar 2022. doi:10.1145/3510449.

38 Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proc. 17th EDBT*, pages 283–294, 2014. doi:10.5441/002/edbt.2014.27.

39 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Proc. 14th SEA*, pages 138–149, 2014. doi:10.1007/978-3-319-07959-2_12.

40 Ely Porat. An optimal Bloom filter replacement based on matrix solving. In *Proc. 4th CSR*, pages 263–273, 2009. doi:10.1007/978-3-642-03351-3_25.

41 Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient Bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009. doi:10.1145/1498698.1594230.

42 Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter. *CoRR*, abs/2006.03176, 2020. URL: https://arxiv.org/abs/2006.03176, arXiv:2006.03176.

43 Stefan Walzer. *Random Hypergraphs for Hashing-Based Data Structures*. PhD thesis, Technische Universität Ilmenau, 2020. URL: https://www.db-thueringen.de/receive/dbt_mods_00047127.

44 Stefan Walzer. Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures. In *Proc. 32nd SODA*, pages 2194–2211. SIAM, 2021. doi:10.1137/1.9781611976465.131.

A Further Experimental Data

The following figures and tables contain

Figures 7 and 8. Performance-overhead trade-off of AMQs for very high false positive rate (≈ 50%) and very low false positive rate (≈ 0.01%) roughly corresponding to performance of 1-bit retrieval and 16-bit retrieval for the retrieval-based AMQs.

Figures 9 to 11. Performance-overhead trade-off of AMQs as in Figure 1, but seperately for positive queries, negative queries and construction.

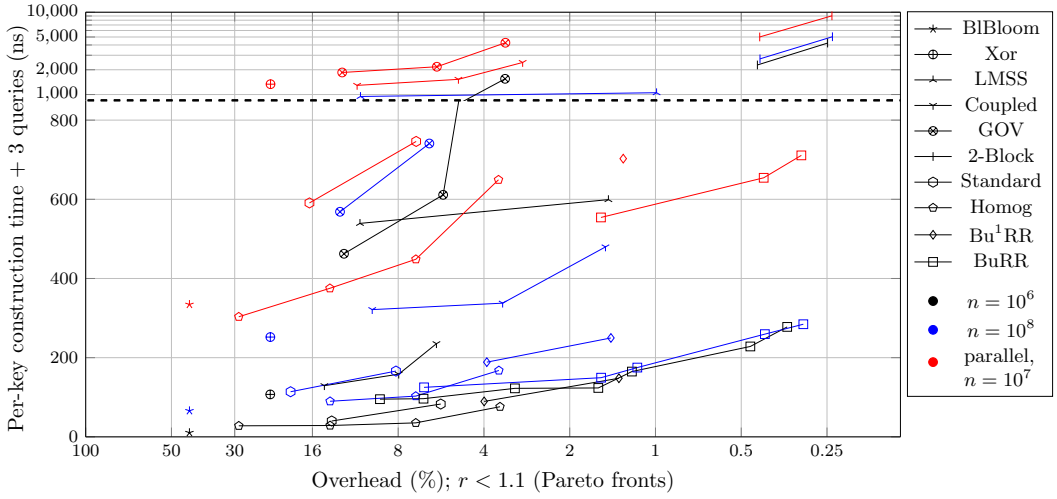

Figure 7 Performance–overhead trade-off for false-positive rate > 46 % for different AMQs and different inputs. This large false-positive rate is the only one for which we have implementations for GOV [28] and 2-block [18]. Note that the vertical axis switches to a logarithmic scale above 900 ns.

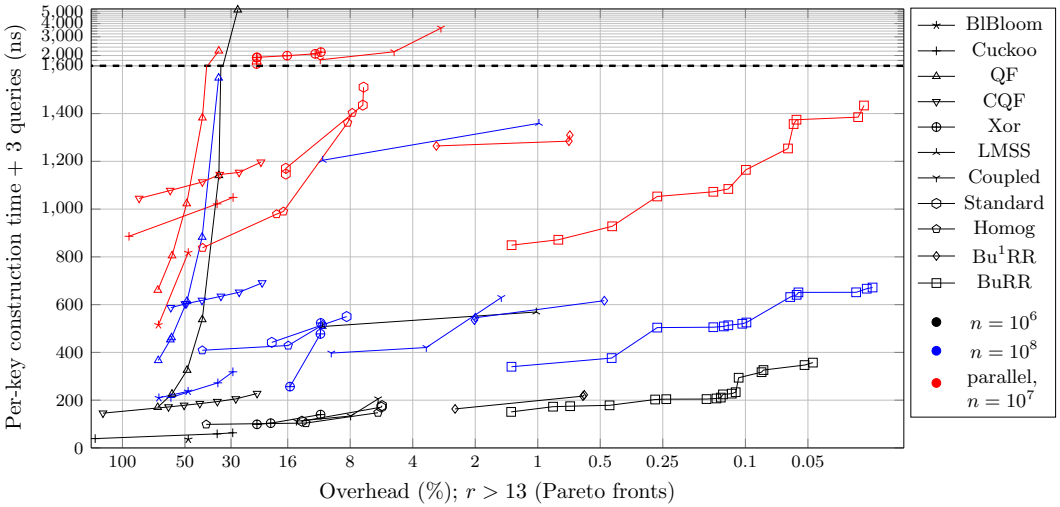

Figure 8 Performance–overhead trade-off for false-positive rate $< 2^{-13} \approx 0.01\,\%$ for different AMQs and different inputs. Logarithmics vertical axis above 1600 ns.

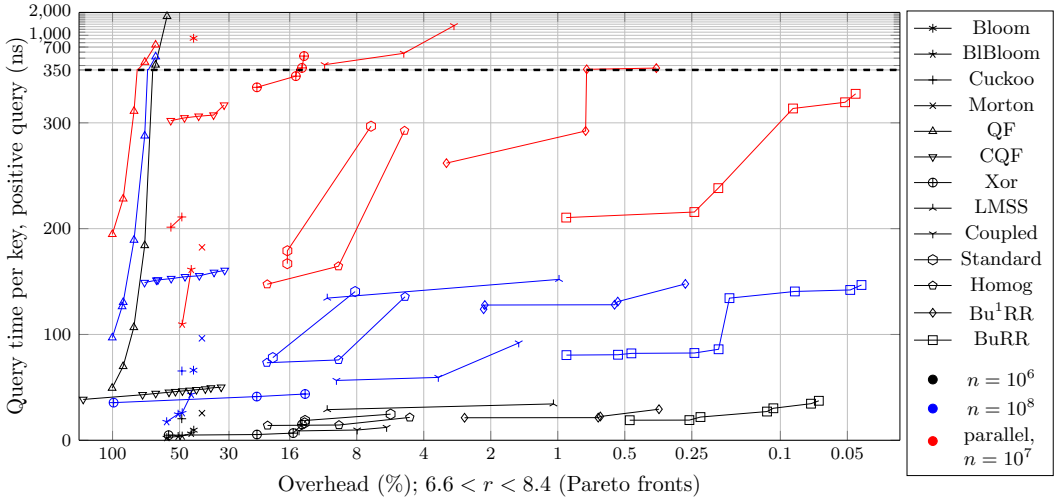

Figure 9 Query time–overhead trade-off for positive queries, false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Note that Xor filters have excellent query time sequentially where random fetches can be performed in parallel but are far from optimal in the parallel setting where the total number of memory accesses matters most. Logarithmic vertical axis above 350 ns.

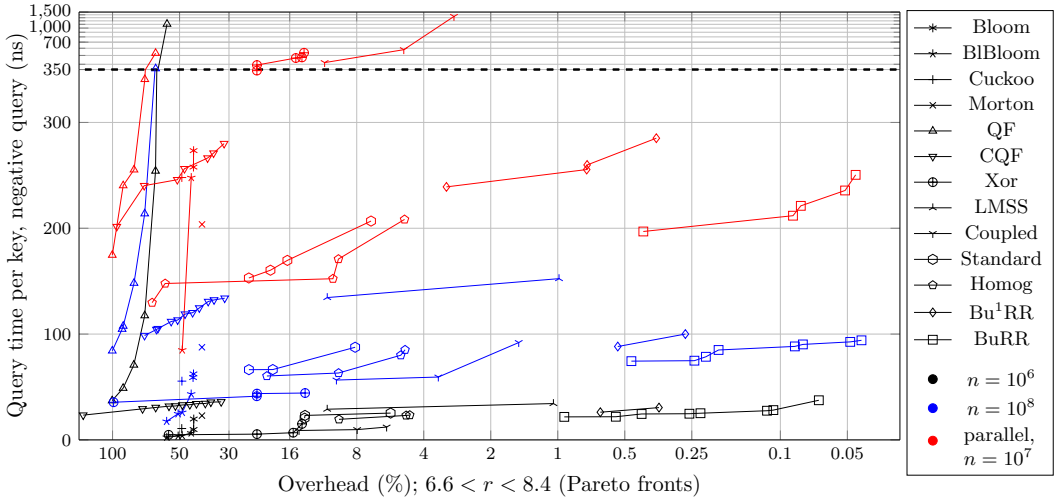

Figure 10 Query time–overhead trade-off for negative queries, false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Again, Xor filters perform well sequentially but suffer in the parallel case. Logarithmic vertical axis above 350 ns.

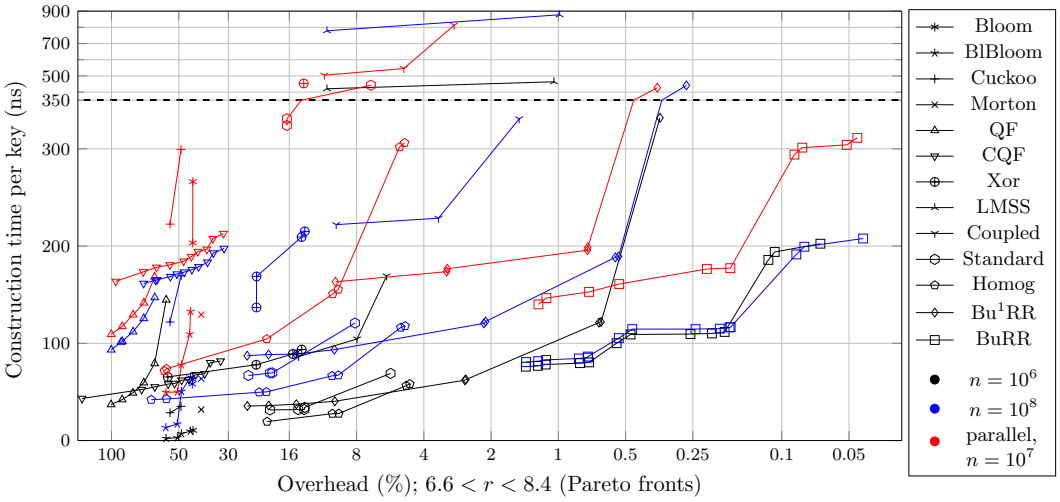

Figure 11 Construction time–overhead trade-off for false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Compressed vertical axis above 350 ns.