The VLDB Journal (2020) 29:731–754
https://doi.org/10.1007/s00778-019-00561-0

SPECIAL ISSUE PAPER

Morton filters: fast, compressed sparse cuckoo filters

Alex D. Breslow[1] · Nuwan S. Jayasena[1]

Received: 1 December 2018 / Revised: 25 July 2019 / Accepted: 29 July 2019 / Published online: 6 August 2019

Abstract
Approximate set membership data structures (ASMDSs) are ubiquitous in computing. They trade a tunable, often small, error rate (ϵ) for large space savings. The canonical ASMDS is the Bloom filter, which supports lookups and insertions but not deletions in its simplest form. Cuckoo filters (CFs), a recently proposed class of ASMDSs, add deletion support and often use fewer bits per item for equal ϵ. This work introduces the Morton filter (MF), a novel CF variant that introduces several key improvements to its progenitor. Like CFs, MFs support lookups, insertions, and deletions, and when using an optional batching interface raise their respective throughputs by up to 2.5×, 20.8×, and 1.3×. MFs achieve these improvements by (1) introducing a compressed block format that permits storing a logically sparse filter compactly in memory, (2) leveraging succinct embedded metadata to prune unnecessary memory accesses, and (3) more heavily biasing insertions to use a single hash function. With these optimizations, lookups, insertions, and deletions often only require accessing a single hardware cache line from the filter. MFs and CFs are then extended to support self-resizing, a feature of quotient filters (another ASMDS that uses fingerprints). MFs self-resize up to 13.9× faster than rank-and-select quotient filters (a state-of-the-art self-resizing filter). These improvements are not at a loss in space efficiency, as MFs typically use comparable to slightly less space than CFs for equal ϵ.

Keywords Morton filter · Cuckoo filter · Hardware cache and memory bandwidth optimized data structures and algorithms · Approximate set membership data structure · Approximate membership query data structure · Rank-and-select

1 Introduction

As time has progressed, systems have added many more levels to the memory hierarchy. In today's enterprise servers, it is not uncommon to have three to four levels of hardware cache, a vast pool of DRAM, several SSDs, and a pool of disks. With each successive level of the hierarchy, latency and bandwidth typically worsen by one or more orders of magnitude. To avoid accessing a slower medium unnecessarily, many applications make use of approximate set membership data structures (ASMDSs). An ASMDS like a set data structure answers set membership queries (i.e., is an item e an element of the set S?). However, unlike a set, which always reports with certitude whether e is in S, an ASMDS may report false positives (i.e., falsely state that e is in S) with a worst-case expected error rate of $0 \leq \epsilon \leq 1$. An ASMDS does not report false negatives [33]: If an ASMDS reports e not in S, then it certainly is not. A core benefit of an ASMDS is that its error rate ϵ is typically independent of the size of the data items that are encoded, so an ASMDS can often reside one or two levels higher in the memory hierarchy than the slow medium to which it filters requests.

The most common ASMDS is the Bloom filter [7], which in its simplest form supports insertions and a LIKELY_CONTAINS lookup primitive. Deletions and counting occurrences of an item are supported via a number of different Bloom filter variants, albeit with an increase in the storage cost (where a 2× to 4× increase is not uncommon [10,34]). Bloom filters have been used in data storage systems such as Google's BigTable [18], distributed analytics platforms like Apache Impala [49], bioinformatics applications like the counting of k-mers during DNA sequencing [60], diverse networking applications [15], and more. One of the pitfalls of the Bloom filter is that its simplest version exhibits poor locality of reference, and more cache-friendly blocked variants are typically less space efficient [15].

✉ Alex D. Breslow
alex.breslow@amd.com

Nuwan S. Jayasena
nuwan.jayasena@amd.com

1 Advanced Micro Devices, Inc., AMD Research, 2485 Augustine Drive, Santa Clara, CA 95054, USA

Consequently, a number of other filters have been proposed, of which two of the most practical are the quotient filter [6,71] and cuckoo filter [33]. Both the quotient filter and the cuckoo filter differ from the Bloom filter in that they store fingerprints: short hashes that each typically have a one-to-one correspondence with an item e that is encoded by the filter as belonging to a set S. Both cuckoo filters and quotient filters support deletions and when filled to high load factors (e.g., a 95% full filter) use less space than an equivalent Bloom filter when the desired false positive rate is less than about 1–3%, the usual case for a wide array of applications.

In this work, we focus on the cuckoo filter (CF) and present a novel variant, the *Morton filter* (MF).[1] Versus stock CFs, MFs particularly excel for workloads that use large filters that are too big to fit in the fast private caches. Since the latency cost of a last level cache (LLC) or memory access is on the order of tens to hundreds of CPU cycles, MFs use that delay to execute additional computation that helps to reduce the required LLC and memory traffic per operation (e.g., lookup, insertion, or deletion). Provided that the mean memory access time is sufficiently long, that extra computation can be almost entirely hidden from the critical path by overlapping it with LLC and memory data transfers.

Like a CF, an MF is logically organized as a linear array of buckets, with each bucket containing a fixed number of slots that can each store a single fingerprint. Fingerprints are mapped to the filter by emplacing the fingerprint in one of two *candidate buckets*, whose indexes are independently determined by two hash functions (H_1 and H_2) that each operate on the key and output a different bucket index. Provided that one candidate bucket has spare capacity, the insertion trivially succeeds. Conflicts where both candidate buckets are full are resolved via cuckoo hashing [70], a hashing technique that triggers a recursive chain of evictions.

Despite these similarities, MFs differ in several key ways. MFs decouple their logical interpretation from how their data are stored in memory. They logically underload the filter and leverage a compressed block format that replaces storage of space-hungry empty slots with a series of *fullness counters* that track the load of each logical bucket. With fullness counters, reads and updates to an MF happen in situ without explicit need for materialization of the logical interpretation of the block. This zero compression makes logically underloading the filter (1) space efficient because many mostly empty buckets can be packed into a single cache block and (2) high performance because accesses occur directly on the compressed representation and only on occupied slots.

With logically underloaded buckets, most insertions only require accessing a single cache line from the MF since H_1 most often succeeds in placing the fingerprint. Thus, most cuckoo hashing is also avoided, and insertion throughput improves by up to 20.8×. Increased biasing in favor of H_1 during insertions also improves lookup throughput, as on subsequent retrieval, the fingerprints placed with H_1 are found in the first bucket, and thus, we can skip accessing the hardware cache block containing the second bucket. For *negative lookups* (queries to keys that were never inserted into the MF), the filter employs an Overflow Tracking Array (OTA), a simple, in-block bit vector that tracks when fingerprints cannot be placed using H_1. By checking the OTA, most negative lookups only require accessing a single bucket, even when the filter is heavily loaded. Thus, regardless of the type of lookup (positive, false positive, or negative), typically only one bucket needs to be accessed. When buckets are cache line resident, most often only one unique cache access is needed per MF probe and at most two, a savings of close to 50%.

Further, an MF typically performs many fewer fingerprint comparisons than a CF, with fewer than one fingerprint comparison per lookup not uncommon, even when the MF is heavily loaded. Instead, many negative lookups are fully or partially resolved simply by examining one or two fullness counters and a bit in the OTA.

Due to this compression, sparsity, biasing, and an interface employing batched lookups and updates, MFs attain improved throughput, space usage, and flexibility. With fewer comparisons and a reduced number of cache accesses, MFs boost lookup, deletion, and insertion throughputs, respectively, by as much as 2.5×, 1.3×, and 20.8× over a stock CF. Similarly, these traits permit using shorter fingerprints because false positives are integrally tied to the number of fingerprint comparisons. Consequently, the space overhead of the fullness counters and OTA can largely be hidden, and space per item can often be reduced by approximately 0.5 to 1.0 bits over a CF with the same ϵ.

MFs further improve space utilization by doing away with the sizing restrictions of their predecessors. Whereas CFs historically have required a power of two number of buckets and quotient filters have used a power of two number of slots, MFs solely require that the number of buckets be a multiple of two. This feature yields further space savings, especially in cases where the number of fingerprints that need to be stored is just over the threshold that would require a larger filter size. The enabler of these advancements is *even-odd partial key cuckoo hashing*, a novel hashing mechanism that uses bucket parity and small, localized odd offsets to map fingerprints between a pair of even and odd buckets. Because offsets are small, many pairs of candidate buckets appear within the same page of virtual memory, which means that a single translation lookaside buffer (TLB) entry can often service virtual to physical memory translations for both buckets.

Aside from performance and space utilization improvements, MFs extend the feature set of CFs to include the self-resizing operation from quotient filters (i.e., resize the filter solely by leveraging its content, not the items that were

[1] Named after a certain elephant's half-bird baby [27].

used to produce it). MFs' ability to self-resize extends their use to dynamic workloads where the required filter size is not known in advance.

Our contributions are as follows:

1. We present the design of the Morton filter, a novel ASMDS that uses compression, sparsity, and biasing to improve throughput without sacrificing on space efficiency or flexibility. MFs improve performance over CFs by making accesses to fewer cache lines during filter reads and updates.
2. We conduct a detailed performance evaluation of the MF on both AMD and Intel server processors.
3. We greatly ameliorate the insertion throughput performance collapse problem for fingerprint-based ASMDSs at high loads ($>100\times$ for a CF) by decoupling an MF's logical representation from how it is stored.
4. We present a fast algorithm for summing fullness counters that is key to an MF's high performance and which can be applied in other contexts.
5. We develop even-odd partial key cuckoo hashing, a hashing mechanism that does away with requiring the total buckets be a power of two and reduces TLB misses by placing many pairs of candidate buckets in the same page of virtual memory.
6. We open-source our C++ MF library at https://github.com/AMDComputeLibraries/morton_filter. Our code's high-performance interface employs batching. However, item-at-time processing is also supported via different APIs.

2 Cuckoo filters

In this section, we describe the cuckoo filter (CF) [33], the ancestor of the MF.

2.1 Baseline design

CFs are hash sets that store *fingerprints*, where each fingerprint is computed by using a hash function H_F, which takes as input a key representing an item in the set and maps it to a fixed-width hash. A CF is structured as a 2D matrix, where rows correspond to fixed-width associative units known as *buckets* and cells within a row to *slots*, with each slot capable of storing a fingerprint. Prior work often uses 4-slot buckets [33].

To map each key's fingerprint to the filter and to largely resolve collisions, Fan et al. encode a key's membership in the set by storing its fingerprint in one of two *candidate buckets*. The key's two candidate buckets are independently computed using two hash functions H_1 and H_2. H_1 takes the key as input and produces the index of one candidate bucket, and H_2 operates on the same key and produces the index of the other [33].

2.2 Insertions

On insertions, provided that at least one of the slots is empty across the two candidate buckets, the operation completes by storing the fingerprint (computed by $H_F(key)$) in one of the empty locations. If no slot is free, *cuckoo hashing* [70] is employed. Cuckoo hashing picks a fingerprint within one of the two buckets, evicts that fingerprint, and stores the new fingerprint in the newly vacated slot. The evicted fingerprint is then rehashed to its alternate bucket using its alternate hash function. To compute the alternate hash function simply by using the bucket index and fingerprint as inputs, they define a new hash function H', which takes the fingerprint and its current bucket as input and returns the other candidate bucket. So, if the fingerprint is currently found in the first candidate bucket given by $H_1(key)$, H' yields the alternate candidate given by $H_2(key)$ and vice versa. If the alternate candidate has a free slot, the evicted fingerprint is written there. In cases where the alternate bucket has no free slot, a fingerprint from that bucket is kicked out, the initial evicted fingerprint takes its place, and the algorithm proceeds recursively on the newly evicted fingerprint until all fingerprints have been relocated or failure is declared.

Two example insertions are shown in Fig. 1. The first is for key K_x, which succeeds because H_2 maps its fingerprint x to a Bucket 0, which has a vacant slot. The second is for key K_y, where it initially fails to find a vacant slot in either candidate bucket (6 and 4), and therefore uses cuckoo hashing to displace a chain of fingerprints beginning with 1011 in Bucket 6 and ending with 1101 in Bucket 2 moving to one of the free slots in Bucket 1. Note that in practice, the series of displacements may occur in reverse order in an opti-

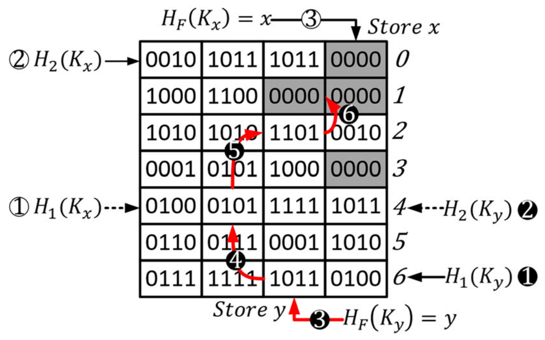


Fig. 1 Insertion of two different keys K_x and K_y into the filter by storing their respective fingerprints x and y. Empty slots are shown in gray. K_x's insertion only involves accessing its two candidate buckets (4 and 0) since 0 has a free slot, but K_y's candidates (6 and 4) are both full, so a series of fingerprints are displaced each to their alternate candidate to make an empty slot for y in Bucket 6. The updated filter is shown in Fig. 2

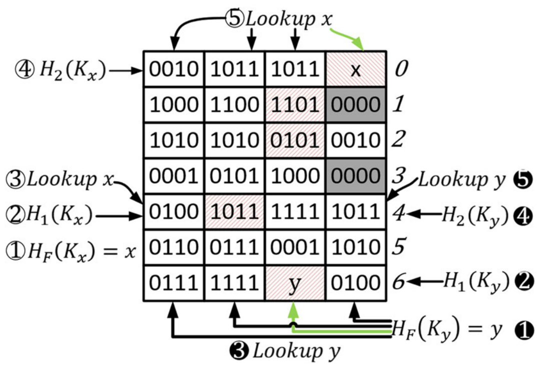


Fig. 2 Lookup of two different keys K_x and K_y following the successful insertion of their respective fingerprints x and y in Fig. 1. For K_y, steps 4 and 5 can optionally be skipped, since y is found in the first candidate bucket

mized implementation to avoid storing displaced fingerprints at each step.

2.3 Lookups

On lookups, the algorithm computes H_1 and H_2 on the key to compute its candidate buckets and H_F to determine its fingerprint. If the fingerprint appears in any of the slots across the two candidate buckets, the lookup returns *LIKELY_IN_SET*, else *NOT_IN_SET*. The certitude of *LIKELY_IN_SET* is subject to an expected error rate ϵ that is tunable by assigning an appropriate bit width to each fingerprint. Increasing the width of each fingerprint by one bit roughly halves ϵ. It is worth noting that the actual incidence of false positives will interpolate with high probability between 0 (all queries are to items inserted in the filter) to roughly ϵ (none of the queried items were inserted in the filter) subject to whether lookups are to true members of the encoded set.

Figure 2 follows the insertion of keys K_x and K_y into the filter in Fig. 1. Even though K_y triggered a series of displacements, since it is only allowed to insert y in one of its two candidate buckets, only Buckets 6 and 4 need to be searched. The same holds for any other queried key: At most, two buckets need to be probed.

2.4 Modeling the error rate and space use

In this section, we present formulae for calculating the error rate of a cuckoo filter, its space usage per item, and show how the insights from this study can be leveraged to design an MF (see Sect. 3 for a high-level MF description). Table 1 provides a glossary of symbols.

A CF has an error rate ϵ which reports the expected ratio of false positives to total LIKELY_CONTAINS queries that are true negatives (i.e., no risk of a false positive on a true positive). To understand how ϵ is calculated given a filter, we first introduce several terms: S the slots per bucket, b the expected number of buckets that need to be searched per negative lookup, α the load factor, and f the bits per fingerprint. When comparing an f-bit fingerprint to a fingerprint stored in a bucket, the occurrence of *aliasing* (i.e., falsely matching on a fingerprint inserted by another key) is $1/2^f$ if all fingerprint values are equally probable. There are 2^f potential values, and only one of those can alias. To compute the net probability of an alias, prior work by Fan et al. [33] observes that there are S slots per bucket, b is fixed at 2 (they always search both buckets), and therefore the rate of aliasing is $\epsilon = 1 - (1 - 1/2^f)^{bS}$, so the necessary f for a target ϵ is roughly $f = log_2(bS/\epsilon)$, which for their parameters of $S = 4$ and $b = 2$ is $f = 3 + log_2(1/\epsilon)$.

Table 1 Glossary of symbols

ϵ	false positive rate
S	slots per bucket
b	buckets searched per negative lookup
α	the load factor
f	fingerprint length in bits
I	bits per item in the filter

However, what this model discounts is the effect of α, that is, if one is careful and clearly marks empty slots (by reserving one of the 2^f states to encode an empty slot), then there is no way that empties can alias when performing a lookup. Marking empties changes the math slightly, to $\epsilon = 1-(1-1/(2^f-1))^{\alpha bS}$, which alters f to approximately

$$f = log_2(\alpha bS/\epsilon) \quad (1)$$

for typical values of f (i.e., $f > 6$). For underloaded filters, it turns out that that extra α term is important because since $0 \le \alpha \le 1$, its logarithm is less than or equal to zero. For instance, filling the filter half full ($\alpha = 0.5$) means that α in the numerator decreases f's required length by $log_2(\alpha = 0.5) = 1$ bit for a target ϵ. Further, this effect is amplified in the savings in bits per item (shown in Eq. 2). With the additional α and a fixed ϵ, $\alpha = 0.5$ would decrease the required bits per item by $log_2(0.5)/0.5 = 2$ bits over Fan et al.'s more pessimistic model.

$$I = f/\alpha = log_2(\alpha bS/\epsilon)/\alpha \quad (2)$$

However, because the α in the denominator of Eq. 2 dwarfs the impact of the α in the numerator, prior work largely ignores this space savings opportunity. In particular, α needs to be close to 1 for a CF to be space competitive with a Bloom filter [7], which largely negates the positive impact of the α in the numerator. To obtain a large value of α (e.g., > 0.95), there are several options for b and S, but practical constraints limit the viable options. For b, a high-performance implementation is limited to selecting 2. A value of $b = 1$ cannot yield a sufficiently high α even for large values of S. $b > 2$ is also undesirable because it results in additional memory traffic (e.g., $b = 3$ triggers an additional memory request per lookup). For S, larger values improve α but at the expense of a worsening error rate given a fixed f. With each additional fingerprint comparison, the likelihood of a false positive increases. In practice, $S = 4$ is the minimum value that permits an α that exceeds 0.95. Larger values of S could be used at the expense of increased bits per item. As we will see in the proceeding sections, our work gets around these limitations. For further analysis of feasible parameters, we point the reader to Erlingsson et al.'s work on cuckoo hash tables (an analogous hash table rather than an approximate hash set) [30], which provides a concise table showing the trade-offs of b and S.

2.5 Bloom filters and relative optimality

Given $b = 2$, $S = 4$, and $\alpha = 0.95$, we examine a CF's relative space and performance optimality. CFs make very efficient use of space. Whereas a Bloom filter uses approximately $log_2(1/\epsilon)/ln(2) \approx 1.44 log_2(1/\epsilon)$ bits per item [15], these parameters for b, S, and α place the bits per item at about $3.08 + 1.05 log_2(1/\epsilon)$, clearly asymptotically better and not too far off from the information theoretic limit of $log_2(1/\epsilon)$ (see Carter et al. [16]). Fan et al. show that the leading constant can be further improved via Bonomi et al.'s semi-sort optimization [11], which sorts fingerprints within a bucket by a fixed-width prefix and then replaces those prefixes with a code word. With 4-bit prefixes, four prefixes can be replaced with a 12-bit code word, a savings of one bit per fingerprint. That reduces the bits per item to $2.03 + 1.05 log_2(1/\epsilon)$, albeit with a performance hit from sorting, compression, and decompression (see Sect. 7).

On the performance front, an ideal filter only requires accessing a single cache line from the filter for each lookup, insertion, or deletion. CFs by contrast often access considerably more data. With lookups and deletions, there are two main approaches: (1) access both buckets each time or (2) skip accessing the second bucket if a matching fingerprint is found in the first one. Fan et al. implement approach (1) for lookups and approach (2) for deletions [32]. Since lookups access both buckets, which with high probability reside in different hardware cache lines, two unique cache lines are accessed from the CF: $2\times$ more than an idealized filter. Deletions fare better; since ASMDSs only permit deletions to items in the filter (otherwise, false negatives are possible), the expected cost in unique cache lines is typically 1 plus the fraction of items that are inserted using H_2. See Ross [78] and Breslow et al. [14] for further associated trade-offs between (1) and (2).

Insertions are often the furthest from optimal. At a high load factor, it can take many cache accesses to insert a single item. In the proceeding sections, we will show how to largely get around these limitations and achieve lookups, insertions, and deletions that typically only access a single cache line from the filter. For a comparative analysis of MFs, which discusses the decoupling of the $log_2(\alpha)$ term in the numerator from the α term in the denominator in Eq. 2, see Sect. 5.

3 Morton filters

This section describes the MF and elaborates on the principal features that differentiate it from a CF.

3.1 Optimizing for the memory hierarchy

The MF is a reengineered CF that is tuned to make more efficient use of cache and memory bandwidth. Today's memory systems move data in coarse-grain units known as cache lines that are typically 64 to 128 bytes. On a load or store instruction, the entire cache line is fetched and brought up through the memory hierarchy to the L1 data cache. Subsequent accesses to words in the same cache line that occur in short sequence (known as *temporal locality* [5,40]) are cheap because they likely *hit* in the high-bandwidth, low-latency L1 data cache. Typical ASMDS workloads are often cache or memory bound because they employ pseudorandom hash functions to set bits or fingerprints within the filter, which limits their data reuse and taxes the comparatively limited bandwidth of lower-level caches (e.g., L3 or L4) and bandwidth to off-chip memory. In contrast to a CF, which optimizes for latency at the expense of performing two random memory accesses per lookup query, an MF probe performs a single cache access most of the time and at most two. In bandwidth-limited scenarios, these efficiency gains correspond to significant speedups (see Sect. 7). We point the interested reader to prior work by Ross [78], Polychroniou and Ross [73], and Breslow et al. [14] for the related discussion of latency and bandwidth trade-offs in hash tables.

3.2 Logical interpretation

Like a CF, the MF maintains a set of buckets and slots, and fingerprints encoding keys are computed using H_F and mapped to one of two candidates using one of H_1 or H_2. Collisions are resolved using a variant of cuckoo hashing (see Sect. 4.2 and Fig. 7).

3.3 Compressed structure: the block store

The MF stores its data in parameterizable units known as blocks. Blocks have a compressed storage format that stores

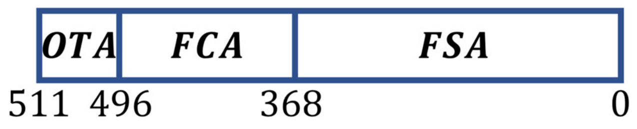


Fig. 3 A sample block in an MF that is performance-optimized for 512-bit cache lines. The block has a 46-slot FSA with 8-bit fingerprints, a 64-slot FCA with 2-bit fullness counters (64 3-slot buckets), and a 16-bit OTA with a bit per slot

both the fingerprints from a fixed number of buckets and accompanying metadata that permits recovering the MF's logical interpretation while solely performing in situ reads and updates to the block. Blocks are stored sequentially in a structure known as the *Block Store*. Block size within the Block Store is dictated by the physical block size of the storage medium for which the MF is optimized. If the filter resides entirely in cache and system memory, then block sizes that evenly divide a cache line are the natural choice (e.g., 256- or 512-bit blocks for 512-bit hardware cache lines). Similarly, block sizes that evenly divide an SSD block are logical choices when the filter is primarily SSD resident.

3.4 Block components

Each MF block has three principal components (shown in Fig. 3), which we detail below:

Fingerprint storage array (FSA)—The FSA is the array that stores the fingerprints from a block. Fingerprints from consecutive buckets within a block are stored one after another in compact, sequential order with no gaps. Empty slots within the FSA are entirely at the end of the buffer. An FSA typically has many fewer slots than the total *logical* slots across all buckets that it stores from the logical interpretation. For instance, in Fig. 3, there are 46 slots for fingerprints in the FSA but a total of $64 * 3 = 192$ slots across the 64 buckets whose fingerprints it stores. Thus, the filter can be logically highly underloaded while allowing the FSA to be mostly full and accordingly conserve space.

Fullness counter array (FCA)—The FCA encodes the logical structure of the block by associating a *fullness counter* with each of its buckets that tracks how many slots are occupied by fingerprints. It enables in situ reads and writes to the serialized buckets in the FSA without the need to materialize a full logical view of the associated block by summing the loads of the buckets prior to the bucket of interest to determine an offset in the FSA where reading should begin. Further, with an FCA, vacant slots in the logical interpretation no longer have to be stored in the FSA, and our implementation uses the FCA to completely skip comparisons to empty fingerprint slots.

Overflow tracking array (OTA)—The OTA in its simplest form is a bit vector that tracks overflows from the block by setting a bit every time a fingerprint overflows (see Sect. 3.8). By querying the OTA, queries determine whether accessing a single bucket is sufficient for correctness or whether both candidates need to be checked.

3.5 Accessing a bucket

A sample block and its logical representation are shown in Fig. 4. In the example, the least significant bit of the block is the farthest to the right. Fullness counters and fingerprints are labeled with the bucket to which they correspond. To show how in situ lookups are possible, we consider the case where we want to examine the contents of Bucket 4. Bucket 4 contains the fingerprints 5, 3, and 6. To determine where to read in the FSA, we can add the loads of Buckets 0 to 3 to provide an offset. These are 1 + 0 + 2 + 1 or 4, so the first fingerprint 5 appears at $FSA[4]$. We know to stop reading at $FSA[6]$ because $FCA[4] = 3$, and so we have already read all three fingerprints. See Sect. 4.1 for a more in-depth description of the algorithm with an accompanying figure.

Note that in the example, the logical interpretation of the block has 18 slots of which a mere 8 are occupied. In a CF, if the block were representative of the average load on the filter, then the bits per item would be $f/(8/18) \approx 2.25f$. However, in the MF, 8 of 10 FSA slots are occupied, so the block's actual load is much higher, and the actual bits per item is $1.25f + FCA\ bits + OTA\ bits$, clearly asymptotically better. Thus, heavily logically underloaded MFs with densely filled $FSAs$ conserve space while allowing inexpensive lookups and updates that are typically localized to a single block and thus a single cache line.

3.6 Primacy

In contrast to a CF, we differentiate between the two hash functions H_1 and H_2. We call H_1 the *primary hash function* and for a given key say that a bucket is its *primary bucket* if its fingerprint would be stored there on an insertion using H_1. We call H_2 the *secondary hash function* and for a given key say that a bucket is its *secondary bucket* if its fingerprint would be stored there on an insertion using H_2. When inserting a fingerprint into an MF, we always first try to place it into its primary bucket and only fall back to the secondary function H_2 when that fails. By heavily biasing insertions in this way, most items in the MF can be found by examining a single bucket, and thus a single hardware cache line.

3.7 Filtering requests to secondary buckets

For *negative lookups* (i.e., where the queried key never had its fingerprint inserted into the table), biasing still helps performance. The OTA tracks all overflows from a block. When H_2 gets used to map a key K's fingerprint to a bucket, we set a bit in the OTA corresponding to the block containing

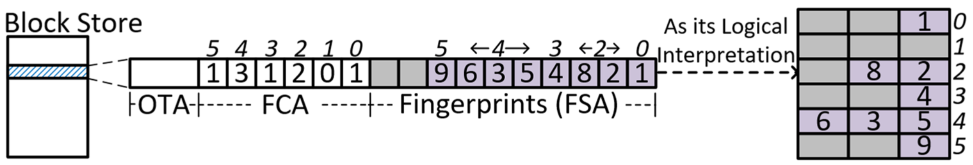


Fig. 4 An MF's Block Store and a sample block's compressed format and logical interpretation, with corresponding buckets labeled 0 to 5. The FCA and FSA state dictates the logical interpretation of the block. Buckets and fingerprints are ordered right to left to be consistent with logical shift operations

K's primary bucket. We select the bit's index by computing a hash function H_{OTA} on K. On a subsequent query to a key K' that was never inserted into the filter but whose primary bucket is in the same block as K's, we compute H_{OTA} on K'. If we hash to an unset bit in the OTA, then the lookup only requires a single bucket access. A set bit requires accessing the other candidate bucket (likely two different cache lines). Bits in the OTA that are previously set remain set on additional overflows that hash to the same bit.

3.8 Types of overflows

At the time of filter initialization, all OTAs across all blocks begin zeroed out. When fingerprints are first inserted, they are inserted exclusively using H_1 and accordingly into their primary buckets. It is only after some time that one of two events will trigger the setting of a bit in the OTA. The first event is a *bucket overflow*. Bucket overflows occur when a key's fingerprint maps to a bucket for which there is no spare logical capacity, that is, when its associated counter in the FCA has already hit its maximum value. The second event is a *block overflow*, which occurs when a key's fingerprint is mapped to a bucket where its block has no spare FSA slots. In both cases, one fingerprint needs to be remapped from the block to make room for the new fingerprint. A bucket overflow requires the evicted fingerprint to come from the new fingerprint's candidate bucket; however, when a block overflow occurs that is not also a bucket overflow, any fingerprint within the block's FSA may be evicted. As it turns out, for most parameter values for the slots per bucket, the vast majority of overflows are purely block overflows. With common fingerprint sizes of several to a few tens of bits, this affords tens of potential fingerprints to evict on any overflow and makes the filter particularly robust during insertions. See Sect. 4.2 for further detail.

3.9 Interplay between buckets and blocks

The addition of the block abstraction is one of the defining features of the MF. By aggregating the loads across the many underloaded buckets that they store, blocks improve the space efficiency of the filter while permitting smaller, less heavily loaded buckets (e.g., a 3-slot bucket with fewer than 1 occupied slot). With small buckets that are mostly empty, most lookups require fewer loads and comparisons and are thus cheaper. For example, an MF that employs the block parameters in Fig. 3 requires fewer than 0.8 fingerprint comparisons per negative LIKELY_CONTAINS query even when 95% of the FSA slots are full, an improvement of more than 10× over a stock CF that checks 8 slots.

Further, small, underloaded buckets afford greater opportunity to batch work from multiple lookups and insertions of multiple items into a shared set of SIMD instructions [36] (see Sect. 4.1 for further discussion).

The large block size greatly benefits insertions. Because the logical interpretation of the filter is sparsely filled, bucket overflows are infrequent because most fullness counters never max out. As such, provided the block has at least one free slot, most insertions are likely to succeed on the first attempt. Thus, overwhelmingly most items are hashed with H_1 (>95% for the parameters in Fig. 3 for FSA occupancies less than or equal to 0.95), so most insertions, deletions, and lookups only access a single cache line from the MF.

3.10 Even-odd partial key cuckoo hashing

The MF employs a different method for calculating H' and H_2 than a stock CF that reduces TLB misses and page faults. We show H_2 and H' below, where K is an arbitrary key, B is the buckets per block, β is the bucket index where K's fingerprint is placed, n is the total buckets, H is a hash function like MurmurHash [3], $map(x, n)$ maps a value x between 0 and $n-1$ inclusive, and $H_F(K)$ is K's fingerprint:

$$\begin{aligned}
H_1(K) &= map(H(K), n)\\
H_2(K) &= map(\ H_1(K) + (-1)^{H_1(K)\&1} * \mathit{offset}(H_F(K)), n)\\
H'(\beta, H_F(K)) &= map(\ \beta + (-1)^{\beta\&1} * \mathit{offset}(H_F(K)), n)\\
\mathit{offset}(F_x) &= [B + (F_x \% \mathit{OFF\_RANGE})] \mid 1
\end{aligned}$$

This formulation logically partitions the filter into two halves: the even buckets and odd buckets. If a bucket is even, then we add an offset to the primary bucket. If it is odd, then we subtract that offset. Offsets are always odd (the |1 term) to enforce switching between partitions. By partitioning in this way, it makes it possible to compute $H'(K)$ and remap K's fingerprint without knowing K itself. This property is also true of Fan et al.'s original scheme, which XORs $H_1(H_F(x))$

with the fingerprint's current bucket to determine its alternate candidate. However, their approach requires the number of buckets in the filter to be a power of two, which in the worst case increases memory use by almost 2×. Our approach does not make this assumption. Rather, we only mandate that the number of buckets be a multiple of two so that H' is invertible.

Our offset calculation has several nice properties. By adding B to the offset, it guarantees that for any key, its two candidate buckets fall in different blocks. This property ensures that rehashing a fingerprint always removes load from the originating block, which is crucial during block overflows. Further, the offset is at most $\pm(B+ OFF\_RANGE)$. Thus, by tuning this value so that it is much less than the number of buckets in a physical memory page, we ensure that most pairs of candidate buckets fall within the same page of memory. This optimization improves both TLB and DRAM row buffer hit ratios, which are crucial for maximizing performance [8,47]. Further, we select an *OFF_RANGE* that is a power of two so that modulo operations can be done with a single logical AND. The $map(x, n)$ primitive is implemented two different ways that both get around performing an integer division. The first method [54] is given by $map(x, n) = (x * n) >> k$, where x is a k-bit integer that is uniformly random between 0 and $2^k - 1$. For the second method, because the offset is bounded, provided that it is smaller than the total buckets in the MF, we do the following:

$$
\begin{aligned}
&if\ x \geq 0 \ \&\&\ x \leq n-1,\ then\ map(x,n) = x\\
&else\ if\ x < 0,\ then\ map(x,n) = x + n\\
&else,\ then\ map(x,n) = x - n
\end{aligned}
$$

4 Algorithms

In this section, we describe the MF's core algorithms.

4.1 Lookups

This section describes how to determine the presence of a key K_x's fingerprint F_x in an MF. A simplified algorithm is presented in Algorithm 1. We first compute the primary hash function H_1 on K_x to determine the global bucket index for its primary bucket (call it $glbi1$) for F_x. Dividing $glbi1$ by the buckets per block B yields the block index. Computing $mod(glbi1, B)$ produces the block-local bucket index $lbi1$. From here, we check for the presence of x in its bucket using *table_read_and_compare*, which performs an in situ check for F_x on K_x's block. No materialization to a logical representation of the block is necessary.

Figure 5 shows *table_read_and_compare* in action. ❶ We first compute K_x's bucket's offset in fingerprints from the start of its primary block. In the example, K_x's lbi is 4, so we sum the loads of all buckets that appear before Bucket 4 (0 through 3 inclusive), which yields an offset (off) of 5 fingerprints. ❷ Since we use zero indexing, that indicates that Bucket 4's first fingerprint appears at index 5. Since the $FCA[lbi] = 2$, that means that Bucket 4 has 2 fingerprints. Therefore, since we begin reading at index 5, we stop reading prior to index $off + FCA[lbi] = 5 + 2 = 7$ (index 6). If any of the fingerprints in the queried range (i.e., 19 or 48) match F_x, then we return true, else false.

Algorithm 1 Algorithm for LIKELY_CONTAINS function

```
1: function LIKELY_CONTAINS($MF$, $K_x$)
2:     $F_x = H_F(K_x)$
3:     $glbi1 = H_1(K_x)$
4:     $block1 = MF.BlockStore[glbi1/B]$
5:     $lbi1 = mod(glbi1, B)$
6:     $match = table\_read\_and\_cmp(block1, lbi1, F_x)$
7:     if ($match$ or $OTA\_bit\_is\_unset(block1, lbi1)$) then
8:         return match
9:     else
10:        $glbi2 = H_2(K_x)$
11:        $block2 = MF.BlockStore[glbi2/B]$
12:        $lbi2 = mod(glbi2, B)$
13:        return $table\_read\_and\_cmp(block2, lbi2, F_x)$
```

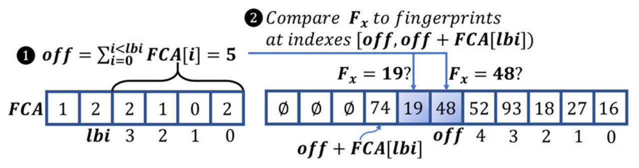

❷ Compare F_x to fingerprints at indexes $[off, off + FCA[lbi])$
❶ $off = \sum_{i=0}^{i<lbi} FCA[i] = 5$
$F_x = 19?$
$F_x = 48?$
FCA
1 2 2 1 0 2
lbi 3 2 1 0
Ø Ø Ø 74 19 48 52 93 18 27 16
$off + FCA[lbi]$
off 4 3 2 1 0

Fig. 5 An example of checking for the presence of fingerprint F_x once the logical bucket index lbi within the block is known

Having returned to Algorithm 1, we then check whether a match was successful or if $lbi1$ maps to an unset bit in the OTA. If either hold, then we return the result. Otherwise, we probe the bucket in the second block and return the result.

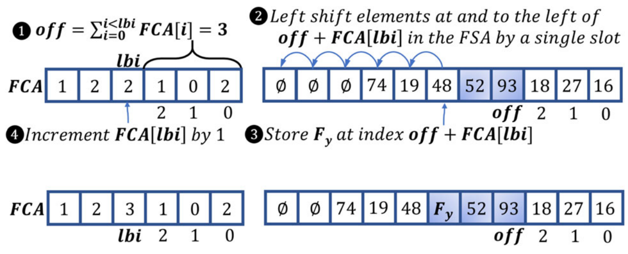
To achieve high performance with this algorithm, we modify it slightly to perform multiple lookups in a flight. All lookups first compute prior to the *if* statement, and we gather those lookups for which the statement evaluated to false, and then perform the *else* statement for the subset where accessing the secondary bucket is necessary. This batching improves performance by permitting improved SIMD vectorization of the code and by reducing the number of branch mispredictions [84]. Batching is akin to the vectorized query processing employed in columnar databases [9] and loop tiling [57,69,94] in high-performance computing.

4.2 Insertions

Algorithm 2 shows the high-level algorithm for insertions. We first attempt to insert into the first candidate bucket at

Algorithm 2 Algorithm for INSERT function

```
1:  function INSERT($MF$, $K_x$)
2:      $F_x = H_F(K_x)$
3:      $glbi1 = H_1(K_x)$
4:      $block1 = MF.BlockStore[glbi1/B]$
5:      $lbi1 = mod(glbi1, B)$
6:      $success = table\_simple\_store(block1, lbi1, F_x)$
7:      if ($success$) then
8:          return success
9:      else
10:         $set\_OTA(block1, lbi1)$
11:         $glbi2 = H_2(K_x)$
12:         $block2 = MF.BlockStore[glbi2/B]$
13:         $lbi2 = mod(glbi2, B)$
14:         $success = table\_simple\_store(block2, lbi2, F_x)$
15:     if ($success$) then
16:         return success
17:     return $res\_conflict(MF, block1, block2, lbi1, lbi2, F_x)$
```

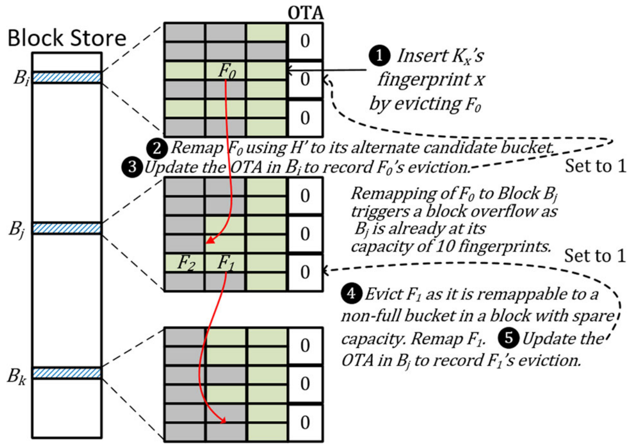


❶ $off = \sum_{i=0}^{i<lbi} FCA[i] = 3$
❷ Left shift elements at and to the left of $off + FCA[lbi]$ in the FSA by a single slot
lbi
FCA 1 2 2 1 0 2
2 1 0
Ø Ø Ø 74 19 48 52 93 18 27 16
off 2 1 0
❹ Increment $FCA[lbi]$ by 1
❸ Store F_y at index $off + FCA[lbi]$
FCA 1 2 3 1 0 2
lbi 2 1 0
Ø Ø 74 19 48 F_y 52 93 18 27 16
off 2 1 0

Fig. 6 An example of inserting a fingerprint F_y into its logical bucket at index lbi within the block. The updated block is shown below. We leave out the OTA for clarity

$glbi1$ (lines 2 through 6). The function $table\_simple\_store$ succeeds if both the candidate bucket and its block's FSA have spare capacity (i.e., no block nor bucket overflow). If $table\_simple\_store$ fails, then the algorithm proceeds to the second candidate bucket (lines 10 through 14). Provided the block and candidate have sufficient capacity, the insertion succeeds. Otherwise, we proceed to the conflict resolution stage (line 17). In this stage, a series of cuckoo hashing displacements are made.

Figure 6 shows the block-local updates that occur during an insertion of a key K_y's fingerprint F_y (the bulk of $table\_simple\_store$). ❶ We begin by computing the bucket offset within the FSA. In this case, K_y's block-local bucket index lbi is 3, so we sum all fullness counters before index 3, which correspond to the loads in fingerprints of the 0th, 1st, and 2nd buckets in the block. ❷ Next, we shift all fingerprints to the left of the end of the bucket (at an offset of $off + FCA[lbi]$) to the left by one slot to vacate a slot for F_y. These operations are inexpensive because many elements are shifted via a single low-latency logical shift instruction, and because Block Store blocks are sized to evenly divide a cache line, only one cache line from the Block Store is accessed per block-level read or update. ❸ F_y is then stored at $FSA[off + FCA[lbi]]$. ❹ The final step is to increment the fullness counter at the bucket's logical index.



Block Store
OTA
B_i
B_j
B_k
F_0
F_2
F_1
0
❶ Insert K_x's fingerprint x by evicting F_0
❷ Remap F_0 using H' to its alternate candidate bucket.
❸ Update the OTA in B_i to record F_0's eviction.
Set to 1
Remapping of F_0 to Block B_j triggers a block overflow as B_j is already at its capacity of 10 fingerprints.
Set to 1
❹ Evict F_1 as it is remappable to a non-full bucket in a block with spare capacity. Remap F_1.
❺ Update the OTA in B_j to record F_1's eviction.

Fig. 7 Insertion of a key K_x into an MF (visualized as the logical interpretation). This insertion is atypical as it requires two levels of evictions to resolve the conflict. In contrast, most insertions only need to update a single block and no cuckoo hashing is necessary, even when blocks are heavily loaded

Figure 7 shows the core functionality of the function $res\_conflict$, with the series of displacements that occur when inserting a sample key K_x's fingerprint x into its primary bucket. ❶ In the example, K_x maps to a bucket that is full (a bucket overflow) within a block that has spare capacity. Cuckoo hashing evicts one of the fingerprints in K_x's candidate (F_0). ❷ F_0 is remapped to its other candidate bucket, which is found in block B_j, and ❸ the OTA in block B_i is updated by setting the bit that H_{OTA} specifies to 1. In the example, blocks have capacity for 10 fingerprints, so B_j is already full even though F_0's other candidate has spare capacity. In this case, B_j experiences a block overflow. In a block overflow without a bucket overflow, any of the existing fingerprints can be evicted to make space for F_0. ❹ In the example, F_1 is evicted from the bucket proceeding F_0's other candidate. F_1 remaps to its alternate candidate, a bucket in B_k, and because B_k is under capacity and F_1's new bucket has a spare slot, the displacement completes. ❺ The OTA in B_j is then updated by setting a bit to record F_1's eviction.

We stress that these complex chains of displacements are infrequent in an MF, contrary to a CF, even at high load factors (e.g., 0.95). With the proper choice of parameters (see Sect. 5), over 95% of items are trivially inserted in their first or second buckets without triggering evictions.

4.3 Deletions

Deletions proceed similarly to lookups. Our example proceeds by deleting the fingerprint F_y that we inserted in Fig. 6. We first compute H_1 on the key (call it K_y) to determine the primary bucket and $H_F(K_y)$ to calculate F_y. From there, we compute the block index and block-local bucket index. The next step is to search the key's primary bucket and delete

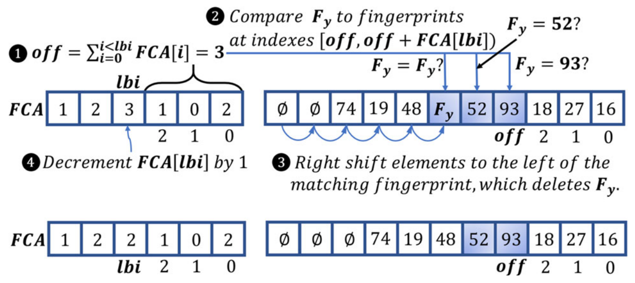

❶ $off = \sum_{i=0}^{i<lbi} FCA[i] = 3$
❷ Compare F_y to fingerprints at indexes $[off, off + FCA[lbi])$
$F_y = 52?$
$F_y = F_y?$
$F_y = 93?$
lbi
FCA
1 2 3 1 0 2
2 1 0
Ø Ø 74 19 48 F_y 52 93 18 27 16
off 2 1 0
❹ Decrement $FCA[lbi]$ by 1
❸ Right shift elements to the left of the matching fingerprint, which deletes F_y.
FCA
1 2 2 1 0 2
lbi 2 1 0
Ø Ø Ø 74 19 48 52 93 18 27 16
off 2 1 0

Fig. 8 An example of checking for the presence of fingerprint F_y once the logical bucket index lbi within the block is known and deleting it on a match. The updated block is shown below. We leave out the OTA for clarity

its fingerprint provided there is a match. Figure 8 shows the block-local operations. ❶ We first sum the fullness counters from index 0 to $lbi - 1$ inclusive, which gives us the offset of the primary bucket's fingerprints within the FSA. ❷ We then perform a comparison between F_y and all the fingerprints in the primary bucket. ❸ On a match, we right shift all fingerprints to the left of the matching fingerprint, which has the effect of deleting the fingerprint. If there are two or more matching fingerprints, we select one and delete it. ❹ Finally, we update the FCA to reflect the primary bucket's new load by decrementing its fullness counter (at index lbi in the FCA) and return.

When the fingerprint is not found in the primary bucket, we calculate $H_2(K_x)$ to produce the secondary bucket's global logical index and proceed as before by computing the block ID and block-local bucket index. We then repeat the process in Fig. 8 and delete a single fingerprint on a match. Note that contrary to lookups, we did not need to check the OTA before proceeding to the secondary bucket. Because ASMDSs only permit deletions to items that have actually been stored in the filter (otherwise false negatives are possible), a failure to match in the primary bucket means the fingerprint must be in the alternate candidate and that the secondary deletion attempt will succeed [33].

Note, our implementation does not clear OTA bits. Repeat insertions and deletions will lead to a growing number of set bits. We combat this effect by biasing block overflows so that they overwhelmingly set the same bits in the OTA by biasing evictions on block overflows from lower order buckets. Given that typically only several percent of fingerprints overflow at load factors at or below 0.95 (less than 5% for the design in Fig. 4), cotuning the OTA's length and the MF's load factor is sufficient for many applications.

For supporting many repeat deletions while sustaining near-maximal load factors (e.g., $\geq$ 0.95), one robust approach is to prepend each fingerprint with a bit that specifies whether the fingerprint is in its primary (or secondary) bucket and force all overflows from a block that map to the same OTA bit to remap to the same alternate block (but very likely different buckets). On deletions of a secondary item x, it is then possible to clear x's corresponding bit in its primary block's OTA if no other fingerprints in its secondary block are simultaneously secondary, would map back to x's originating block, and would hash to the same bit that x would have set in the OTA when it overflowed. A probabilistic variant that saves space by forgoing tagging fingerprints at the expense of not being able to as aggressively clear OTA bits is possible future work.

4.4 Fast reductions for determining bucket boundaries in the FSA

One of the keys to a high-throughput MF is implementing fast reductions on the FCA. Specifically, determining the start of each bucket in the block requires summing all the counts of all fingerprints in buckets that precede it within its block's FSA. A core challenge is that buckets at different indexes require summing differing numbers of fullness counters in the FCA. A naive implementation that issues a variable number of instructions will lead to poor performance. Instead, a branchless algorithm with a fixed instruction sequence is necessary. At first, we considered doing a full exclusive scan (i.e., for every bucket computing the number of fingerprints that precede it). Efficient branchless parallel algorithms like Kogge–Stone [48] exist that require $O(log(n))$ time and map well to the SIMD instruction sets of modern processors (e.g., SSE [76] and AVX [56]).

However, it turns out that a class of more efficient algorithms is possible that consists entirely of simple logic operations (e.g., NOT, AND, and OR), logical shifts, and the *population count* primitive (popcount for short). Popcount is a native instruction on almost every major processor and is a high-throughput, low-latency primitive [65]. Further, high-performance SIMD implementations of popcounts exist that use a combination of lookup tables and permute operations, so even if there is no native popcount instruction, performance-competitive workarounds like Muła et al.'s algorithm and AVX2 implementation are possible [65]. Popcount takes as input a multibyte integer and returns the number of bits that are set to 1. González et al. [39] leverage popcount as part of a high-performance rank-and-select algorithm. Our algorithm generalizes these primitives to arrays of fixed-width counters.

Our approach is shown in Algorithm 3 which, given a bucket at bucket index lbi within the block, computes and returns the number of fingerprints that precede the bucket's fingerprints in the block's FSA.

We first perform a masked copy of the FCA where all counters that appear at lbi or greater are cleared (lines 2-3). This masking ensures that we only count fingerprints that precede the bucket at lbi. In our implementation, this operation also masks out fingerprints that are packed into the same word as the fullness counter array. We next call getPopCount-

Algorithm 3 This algorithm takes as input a block's fullness counter array FCA, a bucket index lbi within the block, the width of each counter in bits w, and then returns the index of the bucket's initial element in the FSA.

```
1: function EXCLUSIVEREDVIAPOPCOUNT($FCA, lbi, w$)
2:     $fullnessCounterArrayMask = (1 << (w * lbi)) - 1$
3:     $mFCA = FCA \& fullnessCounterArrayMask$
4:     $pcMask$ = getPopCountMask($w$)
5:     $sum = 0$
6:     for ($bitPos = 0$; $bitPos < w$; $bitPos++$) do
7:         $sum$ += popCnt($mFCA \& pcMask$) << $bitPos$
8:         $pcMask <<= 1$
       return sum
```

Mask (line 4), which for w-bit fullness counters, returns a mask where the LSB and every wth bit thereafter are set to 1 and the remaining bits to 0. This mask when **and**ed with the masked fullness counter array $mFCA$ selects out all bits that occur in the zeroth position of each of the counters and zeroes out the other digits. For instance, with 4-bit counters, four buckets per block, and hence a 16-bit fullness counter array, the mask $pcMask$ would be **0b0001000100010001** or equivalently **0x1111**. For any value of $bitPos$ between 0 and $w - 1$ inclusive, shifting the $pcMask$ to the left by $bitPos$ selects out the $bitPos$th least significant digit of each w-bit counter (line 8).

The next phase incrementally counts each fingerprint that appears prior to the bucket digit by digit across all remaining counters in the masked copy. We initialize the sum to zero (line 5). During the $bitPos$th pass of the algorithm (line 7), we count the $bitPos$th least significant bit of each counter. That sum is then shifted left by the exponent of the power of two to which the digit corresponds ($bitPos$) before applying it to the growing partial sum sum, which we ultimately return once all passes are complete. Note that loop unrolling will eliminate the branch on line 6.

4.5 Block full array

The *Block Full Array* (BFA) is an optional bit vector that can be paired with an MF to improve insertion throughput at high load factors (e.g., 0.95). A BFA's ith bit is asserted if the ith block in the Block Store's FSA is full. For 64-byte cache lines, the BFA's added storage cost is 0.2%, and thus, it is often small enough to reside in private caches. The BFA is bypassed on lookups. It is queried during insertions to avoid accessing blocks that are full. Insertions and deletions may each trigger updates to the BFA when the FSA goes from being partly full to full and vice versa.

Where the BFA shows the most benefit is during a block overflow. During block overflows, any one of the tens of fingerprints in the FSA can be evicted. The BFA enables quickly selecting a fingerprint whose alternate bucket's block has a spare FSA slot. We thus avoid accessing blocks that would require further subsequent evictions. This optimization significantly reduces unnecessary off-chip data accesses for large filters and accordingly improves throughput by about 30% at a load factor of 0.95 (see Sect. 7.9). The main trade-offs of using the BFA are that it (1) creates added pressure on the hardware caches, (2) makes deletions more expensive by requiring additional accesses to clear BFA bits when block FSAs transition from full to partially empty, and (3) delivers limited benefit for load factors under 0.95.

4.6 Resizing an MF without the source data

In this section, we present an algorithm for self-resizing MFs, which we also backport to CFs (see Sect. 4.8). An MF like a quotient filter supports the ability to resize by a power of two without accessing the source data. This feature is key for applications where there is significant cost to accessing the original data or where it is infeasible because the data no longer exist or their precise location is unknown. In these situations, having a self-resizing filter is useful for adapting to unexpected load or skew. It permits the filter to be sized optimistically to accommodate the common case rather than a seldom-observed, pessimistic tail case that may require significantly oversizing the filter. Small filters take up less space, so operations on them are typically faster because they are often easier to cache closer to the CPU.

When formulating our MF resizing algorithm, our initial inclination was to use the quotienting from quotient filters (QFs). To encode a key's presence into a QF, a hash known as a *signature*[2] is computed where the least significant bits are the fingerprint and the most significant bits specify its preferred index for emplacement. Resizing by factors of two is then trivial by shifting the division between the index and the fingerprint within the signature. For example, with a 28-bit signature (e.g., **0xfe82cd3**) with 16 bits of index (i.e., **0xfe82**) and a 12-bit fingerprint (i.e., **0xcd3**), calculating the new index for the fingerprint within a filter with 16× the capacity simply requires taking the $log_2(16) = 4$ upper bits of the fingerprint (i.e., **0xc**) and repurposing them as the four lower bits of the new index (i.e., **0xfe82c**).

While the approach works well for QFs, MFs have the following additional challenges:

1. A fingerprint's value does not indicate whether it is in its primary or secondary bucket. This information would be necessary to reconstruct the full signature.

[2] Bender et al. [6] use the term *fingerprint* to mean a signature. However, its meaning is different from what a fingerprint is in a CF or MF, as it encompasses both the index and tag, whereas a fingerprint in a CF or MF is just a short hash or tag. We thus use the term *signature* to avoid confusion.

2. We want to preserve an MF's support for non-power of two numbers of buckets. Prior versions of quotienting assume a power of two number of buckets to make the shifting trick work.
3. Preserving the block-level structures like the FCA, FSA, and OTA is non-trivial with resizing.

Our approach handles these hurdles by adding a small amount of state and resizing at block-level boundaries. It maintains a counter R that stores how many times the filter's Block Store's capacity has been doubled. $R = 0$ means the filter's capacity is at its original size. $R = 3$ means that the filter has $8\times$ more buckets than were originally allocated. We leverage R to scale global bucket indexes back and forth between their values in the original and resized Block Stores. See Sect. 4.7 for the precise mechanics.

Like quotienting, R contiguous bits of the original fingerprint are used to direct the computation of the bucket index in the resized filter. These bits could be optionally shifted off to save space or left unmodified when the fingerprint is stored in the new filter. We opt for the latter approach since it allows child blocks to maintain the same format as their parent. Further, our implementation uses C++ templating to enhance compile-time optimization and specialization. The latter approach thus also saves us from needing to instruct the compiler to generate a slew of different code variants for every conceivable set of block layouts that could arise during execution as a result of resizing.

When doubling the capacity, each block in the old Block Store is the sole parent of two child blocks in the new Block Store (i.e., the parent block's fingerprints are split among its two children). The algorithm has several important properties: (1) a fingerprint's block-local bucket indices (one for each candidate bucket) never change on a resizing, (2) an OTA is replicated but never split, and (3) both child blocks are adjacent in the new Block Store, which improves locality (e.g., old block 5 is the parent of new blocks 10 and 11).

Algorithm 4 presents the more general case for increasing an MF's capacity by a nonnegative integral power of two *resize factor* r, and Fig. 9 shows an example of quadrupling the capacity of a filter whose capacity has already doubled (i.e., going from $R = 1$ to $R = 3$). Algorithm 4 begins by allocating a new Block Store with r times more capacity (line 3). It then proceeds to loop over the blocks in the Block Store (line 4) and for every block to copy each of its fingerprints to one of its r child blocks in the new block store (lines 7 to 17). As the algorithm copies the fingerprints to their child blocks, it tracks the number of fingerprints each child block has received up to that point (line 6) and for each fingerprint increments positional pointers in the source block (line 11) and destination child block (line 16). The destination child block is computed by multiplying the parent block ID by r and then adding the child's identifier (the $f - R - log_2(r) - 1$ to $f - R - 1$ least significant bits from the fingerprint F (lines 13–14), with the R correcting for past resizing operations). For each fingerprint that a child block inherits, we increment its bucket's fullness counter in the child block's FCA (line 15). Since a child's block-local bucket index is the same as its parent's, no additional index computation or scaling is necessary. Once a block's fingerprints have been relocated, the OTA is replicated in bulk to each of the child blocks (lines 18–20). Our remapping mechanism guarantees that no complex OTA splitting or scaling is required. Lastly, we deallocate the old Block Store, update the Block Store pointer to point to the new one, and update R.

Algorithm 4 This algorithm takes as input an MF MF, increases its Block Store's capacity by r times, and relocates MF's fingerprints to their new buckets in the resized MF. Note that r must be a power of two.

```
1:  function INCREASECAPACITY($MF, r$)
2:    $oldBS = MF.BlockStore$   // pointer
3:    $newBS = malloc(r * MF.BlockStore.size)$
4:    for ($bID = 0$; $bID < MF.totalBlocks$; $bID++$) do
5:      $fsaIdx = 0$   //Tracks reading in parent FSA
6:      $fsaPtrs[r]\{\}$   //Tracks writing in r child FSAs
7:      for ($lbi = 0$; $lbi < B$; $lbi++$) do
8:        $fullSlots = MF.BlockStore[bID].FCA[lbi]$
9:        for ($sID = 0$; $sID < fullSlots$; $sID++$) do
10:         $F = oldBS[bID].FSA[fsaIdx]$
11:         $fsaIdx++$
12:         $lsbM = r - 1$   //For mod r
13:         $child = (F >> (f - R - log_2(r)))\&lsbM$
14:         $childBID = r * bID + child$
15:         $newBS[childBID].FSA[fsaPtrs[child]] = F$
16:         $fsaPtrs[child]++$
17:         $newBS[childBID].FCA[lbi]++$
18:     for ($child = 0$; $child < r$; $child++$) do
19:       $childBID = r * bID + child$
20:       $newBS[childBID].OTA = oldBS[bID].OTA$
21:   $R += log_2(r)$
22:   $MF.BlockStore = newBS$
23:   $dealloc(oldBS)$
```

4.7 Modifications necessary for supporting resizing

For resizing to properly work, computation of a key's two candidate buckets must factor in the number of times (R) that the MF's capacity has been doubled. For that to work, the offset computation for computing the alternate bucket must be scaled when remapping a fingerprint in a filter that has already been resized. Further, regardless of when a key K's fingerprint x is inserted into the MF (e.g., before or after a resizing), x needs to appear in the correct bucket and block in the current filter. Call x's current bucket $glbi$, which suffices for computing its block index $bi = \lfloor glbi/B \rfloor$ and block-local bucket index $lbi = glbi\%B$.

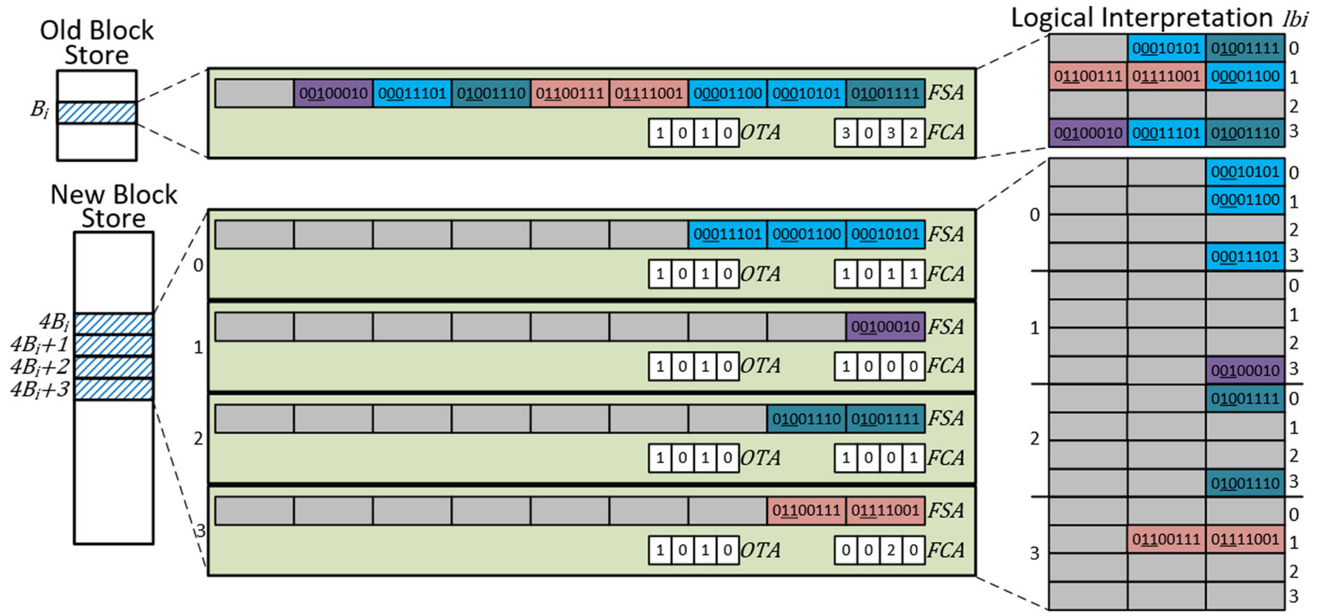


Fig. 9 An example of quadrupling the capacity of an MF whose capacity has already been doubled. A sample block B_i and its child blocks in the new Block Store are shown

To simplify this process, we observe that if we compute what the candidate buckets and blocks would have been in the original Block Store prior to resizing (i.e., when $R = 0$), then we can then subsequently scale them to their correct values in the new Block Store. When calculating the alternate candidate bucket in a filter whose capacity has been doubled R times, we begin by right shifting the bucket's existing block index bi by R. That computation yields the fingerprint's block index were it to be placed in the original filter. Note, in actuality the fingerprint x could have been emplaced after one or more resizings, but bi_0 provides a common point of reference. We then compute the alternate bucket for x in the original filter by plugging in the bucket index in the original filter $glbi_0 = bi_0 * B + lbi$ in for β and x for fingerprint term $(H_F(K) = x)$ in the original H'. That produces the alternate bucket for when $R = 0$. To scale the value in the new filter, lbi is subtracted off and the result is multiplied by 2^R (equivalent to left shifting by R). That yields the global bucket index for the zeroth descendant block's zeroth bucket. To that, we add back in lbi and B times the R most significant bits of x, which places x in the correct bucket of the $(x >> (f - R))\&((1 << R) - 1)$th descendant. The complete set of steps are shown in Eq. 3.

$$\begin{aligned} H'_R(glbi, x) &= ((H'(glbi_0, x) - lbi) << R) \\ &\quad + B * ((x >> (f - R))\&((1 << R) - 1)) + lbi \quad where \\ bi &= \lfloor glbi/B \rfloor \quad //Current\ Block\ Index \\ bi_0 &= bi >> R \quad //Block\ Index\ in\ Original\ MF \\ glbi_0 &= bi_0 * B + lbi \quad //Bucket\ Index\ in\ Orig.\ MF \end{aligned} \tag{3}$$

Equation 4 similarly shows how to compute the new version of H_1 (called H_{1R}) for an MF whose Block Store has 2^R times the initial capacity.

$$\begin{aligned} H_{1R}(K) &= ((H_1(K) - lbi) << R) \\ &\quad + B * ((x >> (f - R))\&((1 << R) - 1)) + lbi \end{aligned} \tag{4}$$

Applying H'_R to H_{1R} yields the new H_2 called H_{2R} (i.e., $H_{2R}(K) = H'_R(H_{1R}(K), x)$).

4.8 Extending self-resizability to cuckoo filters

We backport a variant of the MF's self-resizing algorithm to work with CFs. Since CFs are not typically blocked, we simplify the bucket index computation logic. Unlike an MF, which splits each bucket's fingerprints across two adjacent blocks when doubling capacity, our CF algorithm instead maps a parent's fingerprints to two adjacent child buckets in the new filter. Thus, a fingerprint's child bucket is computed by multiplying its parent bucket's global index by two and adding the value of the Rth most significant bit of the fingerprint. For example, a bucket with index 5 (0b101) in the old filter and fingerprints 0b101011, 0b011100, and 0b010101 with $R = 0$ would map fingerprints 0b$\underline{0}$11100 and 0b$\underline{0}$10101 to a child bucket at index 10 (0b101$\underline{0}$) and 0b$\underline{1}$01011 to a child bucket at index 11 (0b101$\underline{1}$). As with MFs, we architect variants of H_{1R}, H_{2R}, and H'_R that scale the initial outputs of H_1, H_2, and H' to their correct values. A performance comparison with MFs and quotient filters appears in Sect. 7.11. An exhaustive treatment of resizable CFs is the focus of ongoing work.

Table 2 MF glossary of symbols

B	buckets per block
C	the multiplicative slot compression ratio, where $C = 0.25$ corresponds to four slots in the logical interpretation for each physical slot in the FSAs
S	logical slots per bucket
α_L	logical load factor (e.g., $\alpha_L = 0.5$ for 4-slot buckets where on average two slots are full)
α_C	block load factor (e.g., $\alpha_C = 0.8$ for blocks with 40-slot FSAs where on average 35 slots are occupied)
O	number of bits in the OTA of each block
m	expected number of items that overflow a block
b	expected buckets accessed per negative lookup
M	total fingerprints (net total occupied FSA slots)
R	number of times the MF's capacity is doubled

5 Modeling

In this section, we describe a set of models that are used to select the parameters for a filter given a set of constraints. Table 2 lists the parameters that we use in our models.

5.1 False positives and storage costs

The false positive rate ϵ for an MF is given by Eq. 5.

$$\epsilon = 1 - \left(1 - 1/2^{f-R}\right)^{\alpha_L bS} \tag{5}$$

From Eq. 5, we derive Eq. 6: the formula for the fingerprint length f in bits for a target ϵ.

$$f \approx log_2((\alpha_L bS)/\epsilon) + R = log_2((\alpha_C bCS)/\epsilon) + R \tag{6}$$

The bits per item is given by Eq. 7, with the transformed expression obtained by substituting $\alpha_C C$ in for α_L.

$$\begin{aligned} I &= OTA\ bits/item + FCA\ bits/item + FSA\ bits/item \\ &= \frac{O}{\alpha_L BS} + \frac{log_2(S+1)}{\alpha_L S} + \frac{Cf}{\alpha_L} \\ &= \frac{O/(BCS) + log_2(S+1)/(CS) + log_2((\alpha_C bCS)/\epsilon) + R}{\alpha_C} \end{aligned} \tag{7}$$

The OTA bits per item is O divided by the expected occupied fingerprint slots in an FSA ($\alpha_L BS$). The $log_2(S+1)/(\alpha_L S)$ term counts the bits per FCA counter per item in the filter, and the $(Cf)/\alpha_L$ scales the fingerprint length f by the block load factor α_C as $(Cf)/\alpha_L = f/\alpha_C$. For the FSA bits per item term (f/α_C), the α_L in the numerator (i.e., $f = log_2((\alpha_L bS)/\epsilon) + R$) works to reduce f by typically being a small value such that $log_2(\alpha_L)$ shortens f by 1 to 3 bits while the α_C in f/α_C's denominator can be tuned to be close to 1 if the MF's workload is known a priori. These gains are primarily from the FCA and FSA working in concert, which allows us to select α_L to be small (e.g., 0.2) and to shrink S from 4 to 3 or less, all while using comparable or slightly less space than a CF. Further, the OTA helps by reducing b from 2 to close to 1, enough to hide the OTA's space cost while also permitting some space savings.

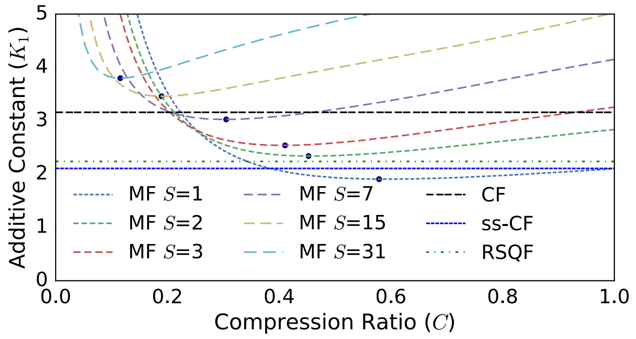


Fig. 10 An MF for some values of C and S uses fewer bits per item than a CF, a CF with semi-sorting (ss-CF), and a rank-and-select quotient filter (RSQF). $\alpha_C = 0.95$ for the MFs and $\alpha = 0.95$ for the RSQF, ss-CF, and CF. For the MFs, we set the block size at 512 bits, and $O = 16$ and $f = 8$

If we fix all parameters except ϵ, I becomes $I = K_1 + K_2 log_2(1/\epsilon)$, where K_1 and K_2 are constants. Figure 10 shows how K_1 varies with the compression ratio C and slots per bucket S for a fixed block load factor α_C of 0.95. We coplot the associated K_1 constants for a $S = 4$ $b = 2$ cuckoo filter (CF), a rank-and-select quotient filter, and a $S = 4$ $b = 2$ cuckoo filter with semi-sort compression (ss-CF) all at a load factor of 0.95. At this design point, $K_2 log_2(1/\epsilon)$ is the same for all filters. The figure demonstrates several key points: (1) MFs use a comparable amount of storage to other filters, (2) there is a fair amount of flexibility when choosing the compression ratio without adversely affecting space usage when the slots per bucket is small, (3) optimizing for space as buckets scale in size requires reducing C, and (4) large buckets may be used with MFs with an additional storage overhead of 1 or 2 bits. This latter point contrasts with CFs, which use an extra bit for each power of two increase in S.

5.2 Lookup costs

An important parameter when modeling the expected cost of each lookup in total buckets is m, the number of items expected to overflow an MF block. Equation 8 presents an approximation for m, which models both block and bucket overflows, as well as their intersection (i.e., an overflow that is both a block and bucket overflow). We ignore cascading evictions due to cuckoo hashing since they are not prevalent (e.g., $<1\%$) for typical parameter values. In our approximation, we model bucket and block overflows using models derived from the Poisson distribution. For $S \geq 2$, block overflows typically overwhelmingly dominate.

$$\begin{aligned} m &\approx bucket\ overflows + block\ overflows \\ &\quad - bucket\ and\ block\ overflows \\ &\approx \alpha_L BS * \Bigg[\frac{\Sigma_{x=S+1}^{M}(x-S)Pr(\alpha_L S, x)}{\alpha_L S} \\ &\quad + \frac{\Sigma_{x=CBS+1}^{M}(x-CBS)Pr(\alpha_L BS, x)}{\alpha_L BS} \\ &\quad - \frac{1}{\alpha_L BS} \sum_{x=CBS+1}^{M} (x-CBS)Pr(\alpha_L BS, x) \\ &\quad * \sum_{y=S+1}^{x} \frac{(y-S)Pr(x/B, y)}{x/B} \Bigg] \\ &\quad where\ Pr(\lambda, x) = \frac{\lambda^x e^{-\lambda}}{x!} \end{aligned} \tag{8}$$

For lookups that are correctly characterized as negatives (excluding false positives), the cost of each such lookup is b. b is one plus the fraction of OTA bits that are set on average for each block. For instance, with a 12-bit OTA per block, if the mean bits that are set is 2, then b would be $1 + 2/12$ (i.e., ≈ 1.167 buckets expected to be accessed per lookup query that returns *false*).

The number of set bits is dependent on the mean fingerprints per block that overflow. These overflows occur both when a bucket does not have sufficient capacity (i.e., when its fullness counter maxes out) and when the block becomes full. Assuming m overflows per block, then Eq. 9 gives the expected negative lookup cost in buckets.

$$\begin{aligned} negative\ lookup\ cost = b &= 1 + 1 - \left(1 - \frac{1}{O}\right)^m \\ &= 2 - \left(\frac{O-1}{O}\right)^m \end{aligned} \tag{9}$$

The final term is the expected fraction of OTA bits that are unset. It is also the probability that a single bit within the OTA is unset. We derive the model by using a balls-into-bins model (see Mitzenmacher and Upfal [64]) where each of the O bits in the OTA is bins and balls are the m overflow items. With m balls thrown uniformly randomly into O bins, the likelihood that an unset bit remains unset after one ball is thrown is $\frac{O-1}{O}$, and we exponentiate by m because the m balls are thrown independently of one another.

For positive lookups (excluding false positives), the lookup cost is shown in Eq. 10 and is approximately one plus the expected fraction of items that overflow a block.

$$positive\ lookup\ cost = 1 + \left(1 - \frac{1}{2^f}\right)^{\alpha_L S} * \frac{m}{\alpha_L BS} \tag{10}$$

$\alpha_L BS$ is the mean occupied fingerprint slots in the FSA per block. The first term in the product corrects for an alias that occurs on the primary bucket when the item's actual fingerprint is in the secondary bucket. Equation 10 is also the expected cost of a deletion since well-formed deletions always succeed (otherwise, false negatives are possible).

The lookup cost for false positives is shown in Eq. 11, which interpolates between 1.5 (the cost of a false positive with a completely full OTA) and 1.0 (the cost of a false positive with an empty OTA). For example, if zeros constitute three quarters of the OTA's bits, then we expect $1.5 - 0.5 * (0.75) = 1.125$ buckets to need to be checked.

$$false\ positive\ lookup\ cost = 1.5 - 0.5 * \left(1 - \frac{1}{O}\right)^m \tag{11}$$

Given P, a ratio of true positives to total lookups, we can compute the expected lookup cost of Eq. 12, the weighted average of the individual lookup costs.

$$\begin{aligned} expected\ lookup\ cost &= P * positive\ lookup\ cost \\ &\quad + (1-P)(1-\epsilon) * negative\ lookup\ cost \\ &\quad + (1-P)(\epsilon) * false\ positive\ lookup\ cost \end{aligned} \tag{12}$$

6 Experimental methodology

We conduct our experiments on an AMD Ryzen$^{\text{TM}}$ Threadripper$^{\text{TM}}$ 1950X processor, which is composed of two 8-core dies for a total of 16 cores, each with 2-way simultaneous multithreading [90,91]. Each core has a 32 KB L1 data cache, a 64 KB L1 instruction cache, a 512 KB L2, and a group of four cores (called a CCX) shares 8 MB of L3 cache (32 MB of L3 in total). We fix the CPU's frequency at 3400 Mhz. Each 8-core die has two memory channels for a total of four [20,83]. The machine has 128 GB of RAM that is clocked at 2133 MHz, and it runs Ubuntu 16.04.4 LTS (Linux 4.11.0). We also perform secondary evaluation on an Intel Skylake-X system.

We compare the MF's throughput to three other filters and implementations from prior work. These are Fan et al.'s cuckoo filter (CF) [32,33], Fan et al.'s CF with semi-sorting (ss-CF) [11,32,33], and Pandey et al.'s rank-and-select quotient filter (RSQF) [71,72]. Fan et al. [33] already demonstrated the CF to be faster than Bloom filters [7], blocked Bloom filters [74],[3] and d-left counting Bloom filters [10,11], so we do not evaluate these designs.

Unless stated otherwise, we run experiments on filters with 128 * 1024 * 1024 slots. We configure the MF to use 8-bit fingerprints and to have a 46-slot FSA, a 128-bit FCA (64 ×

[3] Since publishing our VLDB'18 paper [13], Lang et al. showed that blocked Bloom filters can be faster than CFs [53].

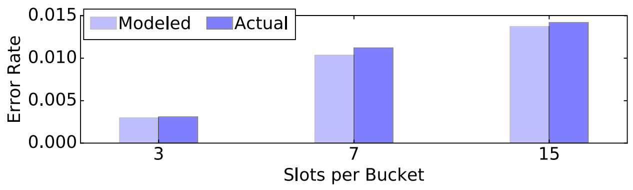


Fig. 11 The MF implementation's false positive rate closely matches Eq. 5. All MFs have a block load factor of 0.95. The MF with 3-slot buckets uses 128 bits for its FCA versus the 7- and 15-slot that use 63 and 64 bits, respectively

2-bit counters), and a 16-bit OTA, for a total of 512 bits per block. It thus contains 64 buckets (each with three logical slots) and has a slot compression ratio (C) of about 0.24. The configuration is the same as the one in Fig. 3 and at a load factor of $\alpha_C = 0.95$ produces an MF that uses 512 / (46 * 0.95) = 11.71 bits per item. We thus compare our implementation to a CF that uses 12-bit fingerprints since Fan et al.'s code does not support 11-bit fingerprints [32,33]. Both filters have roughly equivalent error rates for similar load factors (α_C in the case of the MF).

The 7-slot and 15-slot configurations used in Figs. 11 and 23 also use 8-bit fingerprints. The 7-slot configuration has a 54-slot FSA, a 63-bit FCA (21 $\times$ 3-bit counters), and a 17-bit OTA. The 15-slot configuration has a 54-slot FSA, a 64-bit FCA (16 $\times$ 4-bit counters), and a 16-bit OTA. MF batch size is 128.

We generate 64-bit random integers using C++'s standard library and benchmark the performance of each filter by running separate trials where we fill the filter up to a load factor that is a multiple of 0.05 and then proceed to insert, delete, or look up a number of fingerprints equal to 0.1% of the total slots in the filter. Both the generation of large filters that do not fit into cache and of uniformly random fingerprints are consistent with the evaluations of Fan et al. [33] and Pandey et al. [71]. We rebuild filters after each trial with a new set of random fingerprints to reduce noise. Results are the average of 5 trials. Filters are compiled using g++ (version 5.4.0-6) with the -Ofast -march=native flags, as they yield the best performance. We plot throughput in millions of operations per second (MOPS).

Fan et al.'s implementation packs four 12-bit fingerprints into every 64-bit word and pads the remaining 16 bits. Thus, their CF uses 8 bytes for every 4 fingerprint slots [33]. The MF uses 64 bytes for every 46 fingerprint slots. Thus, the MF is about 186.7 MB in size, whereas the CF is 256 MB (192 MB if it did not pad). The MF also uses a 0.36 MB BFA (Sect. 4.5), which ups its net memory to about 187 MB.

In Sects. 7.3 and 7.6, we present results on the number of buckets accessed per insertion. We attain these results by instrumenting both the MF and CF implementations with access counting that can be enabled at compile time. The counting methodology may double count some accesses if they appear in a cyclic chain of displacements. The counts for MFs are from a revised version of Fan et al.'s random kickout insertion algorithm [33] that has different behaviors depending on the type of overflow (block or bucket).

7 Evaluation

In this section, we present our results, which show our MF implementation to sustain higher throughput than a CF for lookups, insertions, and deletions.

7.1 False positive rate

Figure 11 shows that the false positive rate of MFs closely matches what our models project. The 3-slot design point has a much lower error rate than the 7- and 15-slot variants largely because it allocates more bits per item in the form of metadata (e.g., FCAs). This increased metadata permits the 3-slot design to need fewer fingerprint comparisons per lookup than the 7- and 15-slot designs, which decreases the false positive rate (see Eq. 5). The 7- and 15-slot designs could also be tuned to have lower false positive rates by either using longer fingerprints or by using a logically sparser block design (smaller FSA and larger FCA).

7.2 Lookup throughput

Figure 12 presents the throughput of the MF when 100% of the lookups are true positives. In this configuration, at a load factor of 0.95, a mere 1.05 cache lines from the filter are accessed per query, a reduction of close to 50% over a CF. We observe that at low load factors, the MF is over 2$\times$ faster than a CF and upwards of 1.6$\times$ at high loads.

Figure 13 presents the throughput of the MF when 100% of the lookups are true negatives (a mix of negatives and false positives). The filter achieves a throughput that is 1.3$\times$ to 2.5$\times$ faster. At a load factor of 0.95, the 16-bit OTA has about 11% to 12% of its bits set to 1, so lookups require accessing about 1.11 cache lines (approximately twice as many secondary lookups are required as the positive lookup

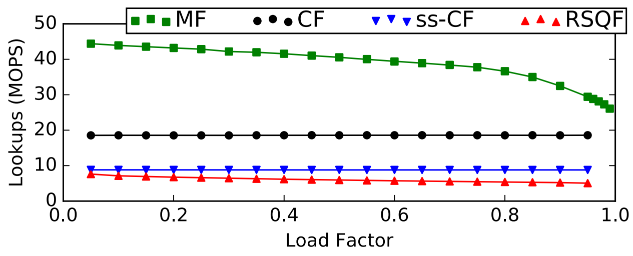


Fig. 12 An MF's positive lookup throughput is about 1.6$\times$ to 2.4$\times$ higher than a CF's

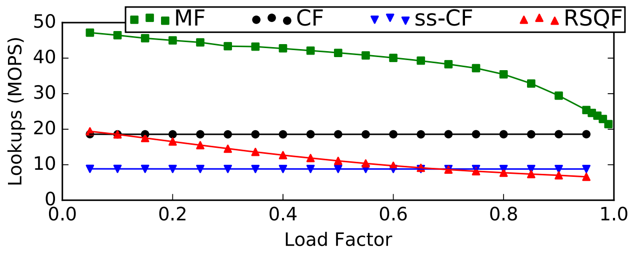


Fig. 13 An MF's negative lookup throughput is about 1.3× to 2.5× higher than a CF's

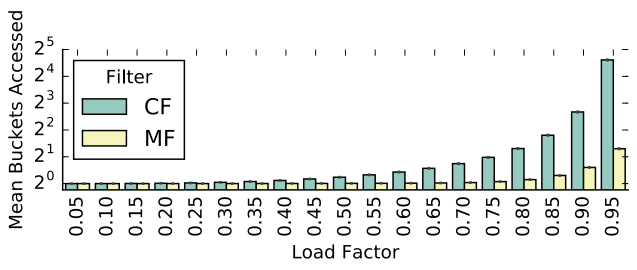


Fig. 15 The number of buckets accessed per insertion for the CF and MF. An MF makes far fewer bucket accesses than a CF (less than one tenth as many at heavy loads)

case). This difference explains why positive lookups sustain higher throughput than negative lookups at heavy loads. To counter this drop in throughput and approximately match the performance of positive lookups, we could double the length of the OTA or reduce the load factor to 0.9.

On an idealized machine and implementation, performance would only drop by about 5% to 11% (i.e., in line with additional data movement). However, it is difficult to achieve that performance in practice due to microarchitectural limitations (e.g., branch mispredictions) and practical trade-offs in software engineering (e.g., trading off between writing code quickly that is cross-platform versus hand-tuning code for a specific processor). Since approximately 90% or more of the lookups never have to perform a secondary access, we focused our efforts on making that fast. An industrial implementation with hand-tuned assembly or vector intrinsics is likely to achieve speedups at high load that are much closer to the reductions in pseudorandom cache accesses.

7.3 Insertion throughput

Like lookups, MF insertion throughput realizes large improvements over CFs for most load factors. Figure 14 demonstrates that an MF is able to sustain high throughput for much longer than a CF. At high loads (e.g., a load factor of 0.75 or higher), the MF is approximately 3× to 20× faster than a comparable CF. This difference matches the simple intuition that buckets with empty slots are easier to insert into than those that are full. Imagine an MF with blocks with 48-slot FSAs and a sample block in which only a single FSA slot is free. Even in this extreme case, provided α_L is low, it is very likely that the block can receive another fingerprint (i.e., very few, if any, of the buckets are likely full). However, if we arrange those same 48 slots into 12 4-slot CF buckets, the probability that we map to a bucket with a free slot is 1/12. Thus, whereas the MF very likely only needs to access one cache line from the filter's storage, the CF is expected to access two or more for 11 out of every 12 insertions, as it needs to access at least one secondary bucket. This disparity is shown quantitatively in Fig. 15, which plots the average number of bucket accesses required per insertion for both CFs and MFs, respectively, as the load factor is varied. At heavy loads, the MF makes several to many times fewer bucket accesses than a comparable MF. We additionally measure the mean bucket accesses to fill an MF and CF from empty to a load factor of 0.95. The MF's reduction in accesses is the key to improving throughput, as cache and memory bandwidth (the typical bottlenecks for an ASMDS) are much more efficiently used. For further validation, see Sect. 7.12.

7.4 Deletion throughput

MF deletion throughput is about 1.1× to 1.3× higher than a CF's (Fig. 16). Like lookups and insertions, the improvement is driven by reducing cache and memory traffic per operation. With the MF parameters of Fig. 3, over 95% of deletions never access more than one MF cache line (even for α_C=0.95).

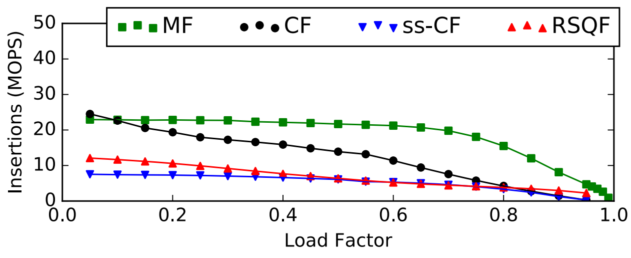


Fig. 14 An MF's insertion throughput is 0.94× to 20.8× that of a CF

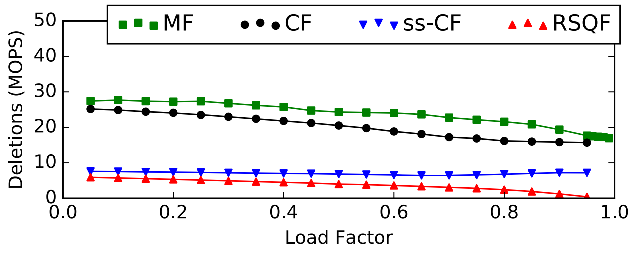


Fig. 16 An MF's deletion throughput is 1.1× to 1.3× higher than that of a CF

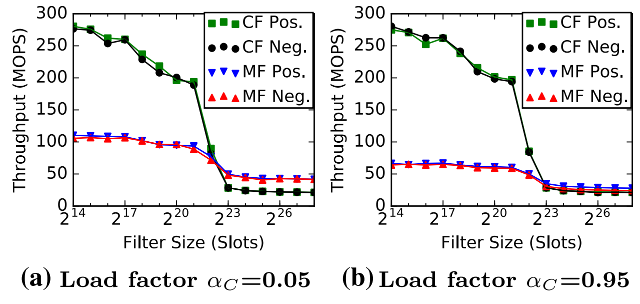


Fig. 17 A comparison of CF and MF lookup throughput as the filter size is varied. MFs surpass CF throughput for filters at 8 * 1024 * 1024 (2^{23}) slots

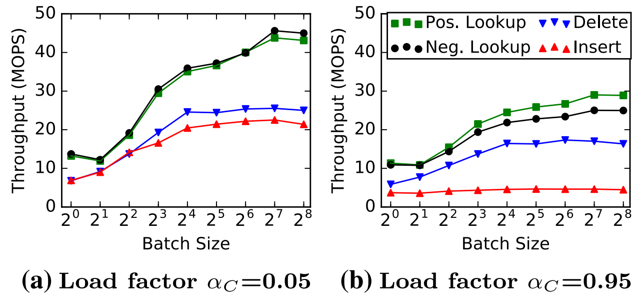


Fig. 19 MF throughput as the batch size is varied. A batch size of 128 (2^7) is the smallest that yields peak throughput

7.5 Sweet spot for Morton filters

Prior sections showed MFs are often considerably faster for large filter sizes. In this section, we quantify the precise point at which MFs outperform CFs. Figure 17a, b presents the comparative lookup throughputs of MFs and CFs as the filter size is varied. CFs are faster at small sizes, but MFs become faster for filters with greater than 2^{23} slots, when the CF is 16 MB in size. Insertions and deletions show similar trends. 16 MB is twice the size of the 8 MB of L3 per CCX (see Sect. 6), or when at most 50% of the CF fits in a CCX's chunk of L3. Future work may explore selectively enabling or disabling MF optimizations subject to the filter size. Smaller filters typically have a lower mean memory access time, which means optimizations to reduce data movement need to execute in fewer instructions and thus be simpler.

7.6 Throughput impact of optimizations

This section explores the vital role that batching and branchless popcount-accelerated reduction optimizations collectively play in yielding a high-performance MF implementation. See Sects. 7.7 and 7.8 for further evaluation of each optimization in isolation. Figure 18 presents the results. Adding batching improves throughput by roughly 2× (except for insertions). Replacing the naive accumulator loop with Algorithm 3 yields another 2× to 3× improvement. Insertion results are the net throughput from filling an MF from empty to a block load factor α_C of 0.95. All other results are the throughput at $\alpha_C = 0.95$.

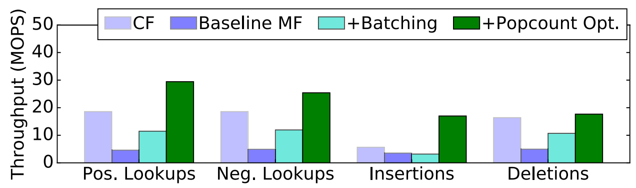


Fig. 18 MF throughput surges with the cumulative addition of optimizations ($\alpha_C = 0.95$)

The optimized MF fills to a load factor of 0.95 about 3× faster than the CF. This speedup principally comes from reducing the mean number of buckets accessed per insertion during construction from 2.08 to 1.08, respectively, for the CF and MF. Part of that 3× increase also comes from improving the tail, which is also critical to performance since tail events can be disproportionately more expensive, particularly since they are likely to cause branch mispredictions. While 2.08/1.08 represents a roughly 2× improvement, the worst-case number of buckets accessed on an insertion reduces from around 370 (CF) to 32 (MF). That more than 90% reduction in bucket accesses is indicative of similar reductions throughout the tail.

7.7 Tuning the batch size

Figure 19a, b shows the throughput of MF operations as the batch size is varied, respectively, for load factors of 0.05 and 0.95. Results for load factors between 0.05 and 0.95 exhibit trends that interpolate between Fig. 19a and b. Deletions and insertions achieve near-peak throughput at a batch size of 16 and then plateau. However, for lookups, a larger batch size of 128 is required to maximize throughput. For high load factors (e.g., 0.95), insertion throughput is not particularly sensitive to the batch size because execution time is dominated by insertion conflict resolution routines (i.e., Fig. 7), which use item-at-a-time processing. Batching this part of the insertion implementation would likely raise throughput and batch size sensitivity.

7.8 Reduction method evaluation

Summing fullness counters to determine offsets into the FSA is a compute-intensive component of each MF operation. In this section, we show that our popcount-based method often yields substantial throughput improvements over both a naive accumulator loop and a tree-based SIMD-within-a-register [35] reduction. Figure 20a–c shows the throughput of the different operations at a load factor (α_C) of 0.05 for the 3-, 7-, and 15-slot MF designs, and Fig. 20d–f for

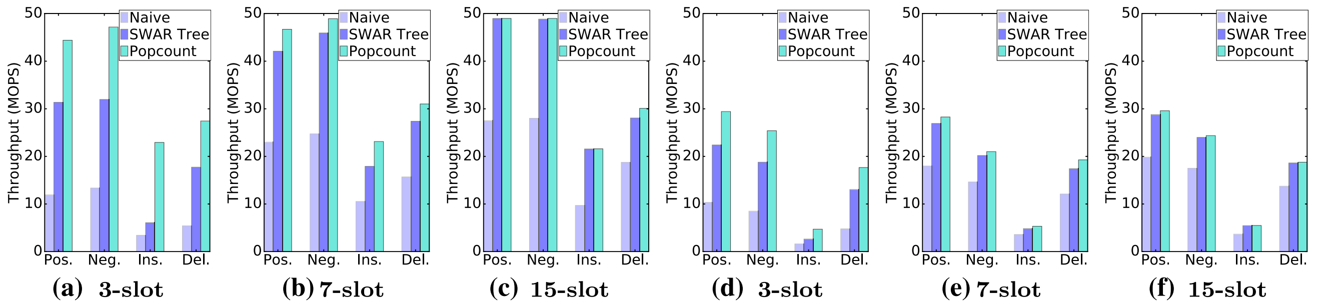


Fig. 20 MF throughput as the summation method that is applied to fullness counters is varied between a naive accumulator loop (**Naive**), a SIMD-within-a-register [35] implementation of a tree-based reduction (**SWAR Tree**), and our popcount-accelerated algorithm (**Popcount**). Our algorithm delivers higher throughput than the other two methods. Subfigures (**a**), (**b**), and (**c**) have a load factor of $\alpha_C = 0.05$. Subfigures (**d**), (**e**), and (**f**) have a load factor of $\alpha_C = 0.95$. **Pos.**, **Neg.**, **Ins.**, and **Del.** correspond to positive lookup, negative lookup, insertion, and deletion throughputs, respectively

$\alpha_C = 0.95$. 3-slot MFs see the largest throughput benefit from our algorithm, followed by the 7-slot design, and then the 15-slot design. Insertions and deletions realize the greatest benefit, with up to more than a 3× insertion throughput uplift for the 3-slot design.

With our algorithm, the 3-slot design only loops over each counter twice ($w = 2$), once per digit. By contrast, the classical tree-based SIMD-within-a-register algorithm requires a number of passes that scales with the log of the number of counters being accumulated (e.g., $log_2(32) = 5$ per 64-bit word of the FCA when $S = 3$). Thus, it performs comparatively better with larger bucket sizes (bigger w values) because there are fewer fullness counters per FCA to accumulate (e.g., a mere 16 buckets in the 15-slot design versus the 64 and 21 buckets, respectively, found in the 3-slot and 7-slot configurations).

7.9 Block full array throughput impact

Figure 21 plots an MF's insertion and deletion throughputs with and without the BFA enabled. Load factors above 0.95 see a large throughput improvement from the BFA. Insertion throughput improves by 1.3× and 2.3× for load factors of 0.95 and 0.99, respectively. However, deletion throughput is reduced by around 10% at low loads and around 20% at heavy loads (e.g., 0.95).

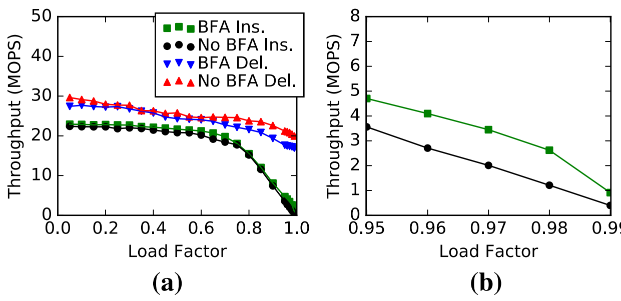


Fig. 21 MF insertion and deletion throughput with and without the BFA enabled. **b** zooms in on the lower right corner of (**a**)

7.10 Flexibility

Figure 22 plots lookup, insertion, and deletion throughput when varying fingerprint length. Despite the wide range in fingerprint lengths, throughput is relatively constant. Figure 23 shows how throughput changes per operation as the

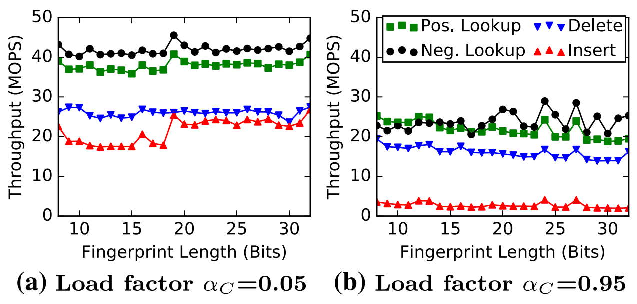


Fig. 22 An MF sustains high throughput for a range of fingerprint lengths ($C \approx 0.24$, $S = 3$, $O \geq 16$)

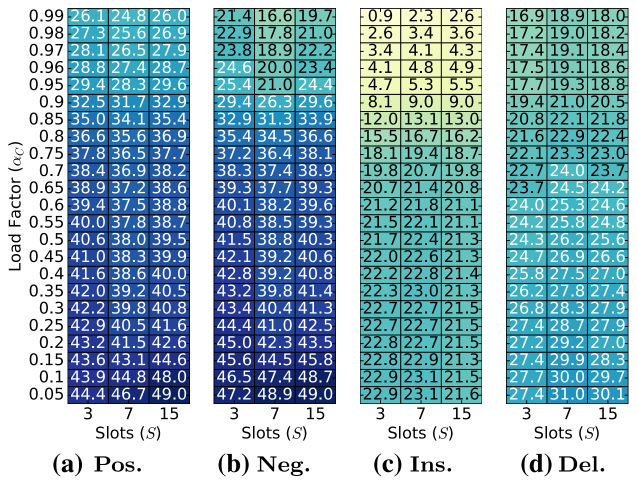


Fig. 23 MF throughput in MOPS for positive lookups (**a**), negative lookups (**b**), insertions (**c**), and deletions (**d**), respectively, as we covary slots per bucket (S) and block load factor (α_C). High throughput is achieved even for large S

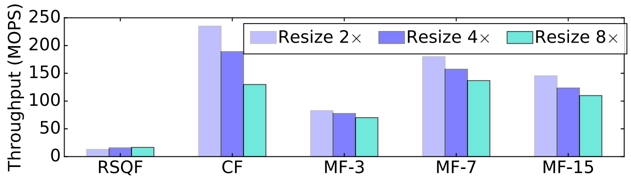
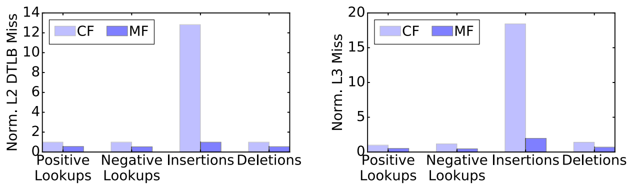


Fig. 24 An MF supports fast resizing. Throughput is in millions of relocated fingerprints per second and includes the cost of memory allocation and deallocation. The CF employs the algorithm from Sect. 4.8



(a) L2 Data TLB Misses (b) L3 Cache Misses

Fig. 25 An MF's L2 data TLB misses and L3 cache misses are lower than a CF's. Results are normalized to a CF executing positive lookups. Lower values are almost always better

slots per bucket is varied. As an MF logically underloads its buckets, the expected number of slots that require checks remains small, and throughput is only modestly affected.

7.11 Resizability

Figure 24 shows that resizing is a comparatively fast operation, even when the MF's capacity is increased by 4× or 8×. For the MFs, resizing is the fastest for the 7-slot configuration. In contrast to the other operations, resizing scans over the Block Store sequentially and therefore exhibits better spatial locality versus lookups, insertions, and deletions that are forced to make many pseudorandom accesses. This speed differential is despite the resizing operation needing to allocate new memory, zero it, and deallocate the old memory. When comparing MF resizing throughput to a backport of the algorithm to CFs (CF in Fig. 24), the MF is a bit slower because it has less predictable branching and does additional work when initializing the OTA and FCA. Compared to the RSQF, MFs achieve approximately 4.2× to 13.9× higher throughput.

Since resizing requires additional computation, we measure its impact on throughput. Enabling resizing for MFs reduces the throughput of each operation by less than 3% for the 3-slot and 15-slot designs. There is minimal impact to throughput since computing scaled block IDs and offsets is comparatively cheaper than the other costs (e.g., random memory accesses and summing counters). Enabling resizing for the 7-slot design is more costly, with throughput overheads of up to 15% for lookups, insertions, and deletions, despite having the highest throughput for the resizing operation.

7.12 Low-level performance metrics

To confirm our intuition that MFs reduce data TLB misses and cache misses per operation (e.g., a lookup), we measure these values with LIKWID [89] a hardware performance counter reading tool and report results in Fig. 25 normalized to a CF's values for positive lookups. Our results show that at a load factor of 0.95, MFs reduce L2 data TLB misses (Fig. 25a) and L3 cache misses per operation (Fig. 25b) by about 50% for both lookups and deletions. During insertions, the MF reduces L2 DTLB and L3 cache misses, respectively, by more than 92% and 89%. These figures are consistent with the order of magnitude improvement in insertion throughput (Fig. 14) and reduction in bucket accesses (Fig. 15).

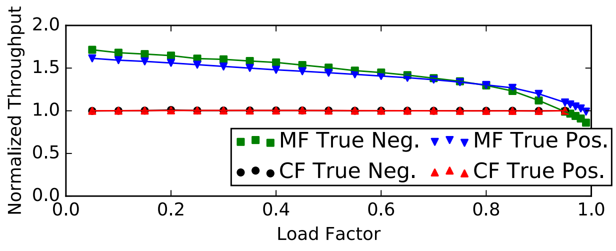
7.13 Cross-platform performance portability

In this section, we show the MF's strong performance portability across different CPU designs by benchmarking on a server with an Intel Skylake-X CPU (hereafter referred to as the Skylake-X server). Since the MF and CF were also the fastest on this platform, we leave out the RSQF and ss-CF, as their throughput as compared to the CF's and MF's did not appreciably change.

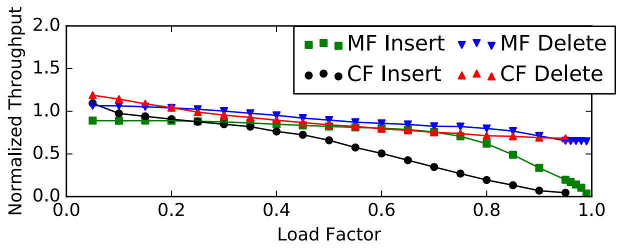
For the Skylake-X server, we perform no additional tuning of our MF implementation, yet despite that, it is still in many cases significantly faster than the CF; the MF's disciplined conservation of cache and memory bandwidth gives it the edge. Given that the CF was developed and performance tuned on an Intel processor like the Skylake-X CPU, it is logical that its performance relative to an MF improves. For lookups (Fig. 26a), the MF attains throughput that matches or exceeds the CF even though it uses fewer bits per item for the same ϵ. MF and CF deletion throughput is roughly equal (Fig. 26b), and MF insertion throughput (Fig. 26b) is higher for load factors at or exceeding 0.5 (up to 4.8× better).

8 Related work

There continues to be significant interest in the design and application of ASMDSs. Bloom first proposed his eponymous filter in 1970 [7]. Since then, it has been used in many different contexts and has evolved into many different variants [1,21,25,34,79,96]. Broder and Mitzenmacher provide a survey of some variants and their applications in computer networking [15]. Fan et al. propose the counting Bloom filter (CBF) for use in SummaryCache [34]. Cohen et al. develop the spectral Bloom filter [21], which like the CBF provides deletions and counting but is more resistant to skew.



(a) Lookup Throughput



(b) Update Throughput

Fig. 26 On a Skylake-X server, MF lookup throughput is on par with to nearly 1.8× higher than a CF's. MF deletion throughput is about 0.90× to 1.1× a CF's. MF insertion throughput is 0.82× to 4.8× that of a CF. Results are normalized to a CF's lookup throughput on a Skylake-X CPU

ASMDS use in databases and data stores—In the database domain, Bloom filters have seen widespread use in accelerating hashed semi-joins (e.g., the Bloomjoin [12,59]). PostgreSQL [85] supports using a Bloom filter as an index, and it is visible as an SQL extension [82].

RocksDB employs Bloom filters and partitioned Bloom filters to speed querying of large data blocks [26]. OceanStore uses a multilayer attenuated Bloom filter [50]. Similar hierarchies of Bloom filters are commonly found in log-structured merge trees [68,80]. LSM trees are employed in a number of data stores such as RocksDB [26], BigTable [18], LevelDB [24], HBase [38], and Cassandra [52].

Fingerprint-based ASMDSs—A number of filters exist that use fingerprints in lieu of setting individual bits. Cuckoo filters [33], *d*-left counting Bloom filters [10], quotient filters [6,71], and TinySet [28] are some examples. TinySet truncates fingerprints to avert overflows that would reduce locality. As such, repeat deletions increase its error rate. *d*-left counting Bloom filters leverage the improved load distribution properties of *d*-left hashing [92] to improve space utilization approximately 2× over a counting Bloom filter.

Compression for sparse matrices—The compression method used by MFs shares similarities with sparse matrix formats like *compressed sparse rows* (CSR) [41,88] and *compressed sparse columns* (CSC) [88] but requires less metadata because positional information within a row (such as in CSR) or within a column (such as in CSC) does not need to be encoded.

Sparse and succinct data structures—Prior works present methods for storing sparse data structures [37,45,67,75,87]. Many use clever hashing or bit vectors with rank and select (see Jacobson [45] and Navarro [66]). Common applications are compressing sparse trees [45,97] or tables [67,87].

Compressed bitmap indices—An MF's compression differs from the typical compressed bitmap algorithms like BBC [2] and the WAH variants which primarily use run length encoding [17,22,44,93,95]. Since fingerprints are about uniformly random, the main compression opportunity is to eliminate storing empty slots, which our simpler approach already does well.

Compression in prior ASMDSs—Other ASMDSs have used compression in the past. Semi-sorting, a form of compression where a portion of each fingerprint is compressed, has been proposed in the context of *d*-left counting Bloom filters [10,11] and cuckoo filters [33]. Semi-sorting could be added to an MF to save additional space. We chose not to use semi-sorting because the gains in space come at significant cost to performance. Further, our implementation would have required additional complexity since we would have had to support encoding and decoding for differing numbers of occupied slots per bucket since empty slots are not explicitly stored. Mitzenmacher provides detailed analysis of design trade-offs when compressing Bloom filters and discusses its applicability to web caching [61].

Cuckoo hash tables—A CF is highly related to cuckoo hash tables (CHTs) [70] and variants [14,78,86]. Rather than storing fingerprints, a CHT stores key-value pairs. Like CFs, CHTs typically have two candidate buckets with four or eight slots [14,30,31,55,78,98]. A commonality of a baseline CF and CHT is that as the load on the table increases, insertion throughput decreases due to a rapid increase in the prevalence and mean length of cuckoo evictions. Prior work addresses this reduction in throughput in several ways. Li et al. employ a breadth-first search (BFS) to reduce the maximum chain length to one that is logarithmic in the maximum number of slots that are checked before declaring failure [55]. Their concurrent CHT outperforms MemC3's [31] that uses 2-choice hashing [4,62] with the random kickout algorithm employed in Fan et al.'s CF [33]. A concurrent MF would likely similarly benefit from using BFS. Sun et al. [86] add metadata that explicitly tracks the graph-theoretic state of the table to prune the search space for cuckoo hashing. Our design avoids this complexity by using blocks that support storing tens of fingerprints. Horton tables [14] convert the final slot of buckets that overflow into a *remap entry array* (REA) that enables lookups that access close to a single bucket, provides many bucket candidates (e.g., 8), and keeps a worst-case lookup cost of two buckets. We implement the OTA as a bit vector rather than an REA for simplicity.

Cuckoo filter adoption and variants—CFs have seen adoption in the areas of key-value stores [77], networking [42,51], and security applications [23,46]. In addition to MFs, a growing body of work proposes design variants and enhancements to CFs [13,19,58,63,77]. Adaptive cuckoo filters [63] employ multiple tables, each with a different fingerprint length and a backing hash table. For items for which there are frequent false positives, the hash table is used to create a longer fingerprint that is inserted into another table. It shares structural similarities with TAGE branch predictors [81]. An adaptive MF could also be created. Dynamic cuckoo filters [19] (inspired by dynamic Bloom filters [43]) chain several cuckoo filters together. When the current CF reaches capacity, another CF is appended to the chain. While this approach allows increasing the capacity, it often requires examining multiple filters during lookups and deletions. Our resizing operation by contrast requires no changes to the lookup algorithm, which typically accesses one bucket and at most two. Position-aware cuckoo filters [51] add a counter per bucket that tracks the number of fingerprints that are hashed by H_1. For each bucket, it first stores all the fingerprints hashed by H_1 and then those hashed there by H_2, which reduces the error rate on lookups. MFs can attain similar false positive rates for the same number of bits and also examine fewer fingerprints and buckets. 2–3 cuckoo filters [29] place two fingerprints in two of three candidate buckets, and in doing so often require more unique cache accesses per lookup than a baseline CF or MF. Lang et al. present techniques to optimize Bloom filter and CF lookup throughput on modern CPUs using SIMD instructions and provide guidance on when each should be used [53].

9 Conclusion

We have presented a high-throughput filter that supports improved throughput for lookups, insertions, and deletions without increasing memory usage. Perhaps most notable is that an MF's insertion throughput is about $3\times$ to $20\times$ higher than a comparable CF for load factors at or above 0.75. Further, lookup and deletion throughput are up to $2.5\times$ and $1.3\times$ faster, respectively. These properties are achieved while also using comparable or fewer bits per item than a CF for a target false positive rate. Key to these advances is the block abstraction and its compressed format, which allows for hiding the storage cost of additional metadata structures by logically underloading the filter and using smaller buckets. The OTA further decreases these costs by reducing aliasing that would require increasing the length of fingerprints by filtering out unnecessary accesses to secondary buckets. With the OTA and a reduction in bucket overflows due to packing many underloaded buckets into a single cache line, lookups most often only have to access a single bucket (one hardware cache line) even when the filter is heavily loaded. We look forward to applying the MF in a variety of contexts due to its memory friendliness. Further, the innovations of this work like the compression and performance optimizations can be applied to a broad range of other data structures such as hash tables (e.g., a cuckoo hash table), various fingerprint-based filters, and algorithms that employ reductions or scans on fixed-width narrow fields or counters.

Acknowledgements We thank the VLDB reviewers and our kind colleagues Shaizeen Aga, Joseph L. Greathouse, Mike Ignatowski, and Gabriel Loh for their time and superb feedback which substantially improved the paper's clarity and quality. We also thank John Kalamatianos and Jagadish Kotra for giving us access to the Skylake-X server and Karen Prairie for her edits. We finally thank the AMD Open Source Review Board, Alan Lee, Chip Freitag, Mike Chu, and the dozens of others who were involved in the auditing and open-sourcing of our Morton filter implementation. AMD is a trademark of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

1. Almeida, P.S., Baquero, C., Preguiça, N.M., Hutchison, D.: Scalable Bloom filters. Inf. Process. Lett. **101**(6), 255–261 (2007)
2. Antoshenkov, G.: Byte-aligned bitmap compression. In DCC, pp. 476 (1995)
3. Appleby, A.: MurmurHash. https://sites.google.com/site/murmurhash (2008). Accessed 2 May 2018
4. Azar, Y., Broder, A.Z., Karlin, A.R., Upfal, E.: Balanced allocations. SIAM J. Comput. **29**(1), 180–200 (1999)
5. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. **5**(2), 78–101 (1966)
6. Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., Zadok, E.: Don't thrash: how to cache your hash on flash. PVLDB **5**(11), 1627–1637 (2012)
7. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. CACM **13**(7), 422–426 (1970)
8. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: memory access. In VLDB, pp. 54–65 (1999)
9. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: hyper-pipelining query execution. In CIDR, pp. 225–237 (2005)
10. Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., Varghese, G.: An improved construction for counting Bloom filters. ESA **6**, 684–695 (2006)
11. Bonomi, F., Mitzenmacher, M., Panigraphy, R., Singh, S., Varghese, G.: Bloom filters via d-left hashing and dynamic bit reassignment extended abstract. In Allerton, pp. 877–883 (2006)
12. Bratbergsengen, K.: Hashing methods and relational algebra operations. In VLDB, pp. 323–333 (1984)
13. Breslow, A., Jayasena, N.: Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. PVLDB **11**(9), 1041–1055 (2018)
14. Breslow, A.D., Zhang, D.P., Greathouse, J.L., Jayasena, N., Tullsen, D.M.: Horton tables: fast hash tables for in-memory data-intensive computing. In USENIX ATC, pp. 281–294 (2016)
15. Broder, A.Z., Mitzenmacher, M.: Network applications of Bloom filters: a survey. Internet Math. **1**(4), 485–509 (2003)
16. Carter, L., Floyd, R., Gill, J., Markowsky, G., Wegman, M.: Exact and approximate membership testers. In STOC, pp. 59–65, New York, NY (1978)
17. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with Roaring bitmaps. Softw. Pract. Exp. **46**(5), 709–719 (2016)

18. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: BigTable: a distributed storage system for structured data. TOCS **26**(2), 4 (2008)
19. Chen, H., Liao, L., Jin, H., Wu, J.: The dynamic cuckoo filter. In ICNP, pp. 1–10 (2017)
20. Clark, M.: A new x86 core architecture for the next generation of computing. In Hot Chips, pp. 1–19 (2016)
21. Cohen, S., Matias, Y.: Spectral Bloom filters. In SIGMOD, pp. 241–252 (2003)
22. Colantonio, A., Pietro, R.D.: Concise: compressed 'n' composable integer set. Inf. Process. Lett. **110**(16), 644–650 (2010)
23. Cui, J., Zhang, J., Zhong, H., Xu, Y.: SPACF: a secure privacy-preserving authentication scheme for VANET with cuckoo filter. IEEE Trans. Veh. Technol. **66**(11), 10283–10295 (2017)
24. Dean, J., Ghemawat, S.: LevelDB: a fast persistent key-value store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, July 27, 2011. Accessed 25 Jan 2017
25. Deng, F., Rafiei, D.: Approximately detecting duplicates for streaming data using Stable Bloom filters. In SIGMOD, pp. 25–36 (2006)
26. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M.: Optimizing space amplification in RocksDB. In CIDR (2017)
27. Dr. Seuss. Horton Hatches the Egg. Random House (1940)
28. Einziger, G., Friedman, R.: TinySet - an access efficient self adjusting Bloom filter construction. TON **25**(4), 2295–2307 (2017)
29. Eppstein, D., Goodrich, M.T., Mitzenmacher, M., Torres, M.R.: 2-3 cuckoo filters for faster triangle listing and set intersection. In PODS, pp. 247–260 (2017)
30. Erlingsson, U., Manasse, M., McSherry, F.: A cool and practical alternative to traditional hash tables. In WDAS (2006)
31. Fan, B., Andersen, D.G., Kaminsky, M.: MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In NSDI, pp. 371–384 (2013)
32. Fan, B., Andersen, D.G., Kaminsky, M.: Cuckoo filter. https://github.com/efficient/cuckoofilter, (2017). Accessed 19 Nov 2017
33. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.: Cuckoo filter: practically better than Bloom. In CoNEXT, pp. 75–88 (2014)
34. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary Cache: a scalable wide-area web cache sharing protocol. TON **8**(3), 281–293 (2000)
35. Fisher, R.J., Dietz, H.G.: Compiling for SIMD within a register. In LCPC, pp. 290–304 (1998)
36. Flynn, M.J.: Some computer organizations and their effectiveness. TOC 21(9):948–960 (1972)
37. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
38. HBase, L George: The Definitive Guide: Random Access to Your Planet-size Data. O'Reilly Media, Inc., New York (2011)
39. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In WEA, pp. 27–38 (2005)
40. Goodman, J.R.: Using cache memory to reduce processor-memory traffic. In ISCA, pp. 124–131 (1983)
41. Greathouse, J.L., Daga, M.: Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In SC, pp. 769–780 (2014)
42. Grissa, M., Yavuz, A.A., Hamdaoui, B.: Cuckoo filter-based location-privacy preservation in database-driven cognitive radio networks. In WSCNIS, pp. 1–7 (2015)
43. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic Bloom filters. TKDE **22**(1), 120–133 (2010)
44. Guzun, G., Canahuate, G., Chiu, D., Sawin, J.: A tunable compression framework for bitmap indices. In ICDE, pp. 484–495 (2014)
45. Jacobson, G.: Space-efficient static trees and graphs. In FOCS, pp. 549–554 (1989)
46. Kales, D., Rechberger, C., Schneider, T., Senker, M., Weinert, C.: Mobile private contact discovery at scale. In USENIX Security (2019)
47. Kandemir, M., Zhao, H., Tang, X., Karakoy, M.: Memory row reuse distance and its role in optimizing application performance. In SIGMETRICS, pp. 137–149 (2015)
48. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. TOC **100**(8), 786–793 (1973)
49. Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., Russell, J., Tsirogiannis, D., Wanderman-Milne, S., Yoder, M.: Impala: a modern, open-source SQL engine for Hadoop. In CIDR, (2015)
50. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S.E., Eaton, P.R., Geels, D., Gummadi, R., Rhea, S.C., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.Y.: . OceanStore: an architecture for global-scale persistent storage. In ASPLOS, pp. 190–201 (2000)
51. Kwon, M., Shankar, V., Reviriego, P.: Position-aware cuckoo filters. In ANCS, pp. 151–153 (2018)
52. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. OSR **44**(2), 35–40 (2010)
53. lang, H., Neumann, T., Kemper, A., Boncz, P.: Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. PVLDB **12**, 502–515 (2019)
54. Lemire, D.: A fast alternative to the modulo reduction. https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/, June 27, (2016). Accessed 07 Jan 2017
55. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In EuroSys, vol 27, pp. 1–27:14 (2014)
56. Lomont, C.: Introduction to Intel advanced vector extensions. Intel White Paper, pp. 1–21 (2011)
57. Loveman, D.B.: Program improvement by source-to-source transformation. J. ACM **24**(1), 121–145 (1977)
58. Luo, L., Guo, D., Rottenstreich, O., Ma, R.T., Luo, X., Ren, B.: The consistent cuckoo filter. In Infocom, (2019)
59. Mackert, L.F., Lohman, G.M.: R* optimizer validation and performance evaluation for distributed queries. In VLDB, pp. 149–159 (1986)
60. Melsted, P., Pritchard, J.K.: Efficient counting of k-mers in DNA sequences using a Bloom filter. BMC Bioinformatics **12**, 333 (2011)
61. Mitzenmacher, M.: Compressed Bloom filters. In PODC, pp. 144–150 (2001)
62. Mitzenmacher, M.: The power of two choices in randomized load balancing. TPDPS **12**(10), 1094–1104 (2001)
63. Mitzenmacher, M., Pontarelli, S., Reviriego, P.: Adaptive cuckoo filters. In ALENEX, pp 36–47
64. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis. Cambridge University Press, Cambridge (2017)
65. Mula, W., Kurz, N., Lemire, D.: Faster population counts using AVX2 instructions. Comput. J. **61**(1), 111–120 (2018)
66. Navarro, G.: Compact Data Structures: A Practical Approach. Cambridge University Press, Cambridge (2016)
67. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In Meeting on Algorithm Engineering & Expermiments, pp 60–70, (2007)
68. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The Log-Structured Merge-tree (LSM-tree). Acta Inform. **33**(4), 351–385 (1996)

69. Padua, D.A., Wolfe, M.J.: Advanced compiler optimizations for supercomputers. CACM **29**(12), 1184–1201 (1986)
70. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
71. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: making every bit count. In SIGMOD, pp. 775–787 (2017)
72. Pandey, P., Johnson, R.: A general-purpose counting filter: counting quotient filter. https://github.com/splatlab/cqf, (2017). Accessed 11 Sep 2017
73. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In SIGMOD, pp. 1493–1508 (2015)
74. Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient Bloom filters. JEA, 14 (2009)
75. Raman, R. Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In SODA, pp. 233–242 (2002)
76. Raman, S.K., Pentkovski, V., Keshava, J.: Implementing streaming SIMD extensions on the Pentium III Processor. IEEE Micro **20**(4), 47–57 (2000)
77. Ren, K., Zheng, Q., Arulraj, J., Gibson, G.: SlimDB: a space-efficient key-value storage engine for semi-sorted data. PVLDB **10**(13), 2037–2048 (2017)
78. Ross, K.A.: Efficient hash probes on modern processors. In Chirkova, R., Dogac, A., Özsu, M.T., Sellis, T.K. (eds), ICDE, pp.1297–1301 (2007)
79. Rottenstreich, O., Kanizo, Y., Keslassy, I.: The variable-increment counting Bloom filter. TON **22**(4), 1092–1105 (2014)
80. Sears, R., Ramakrishnan, R.: bLSM: a general purpose Log Structured Merge tree. In SIGMOD, pp. 217–228 (2012)
81. Seznec, A.: A new case for the TAGE branch predictor. In MICRO, pp. 117–127 (2011)
82. Sigaev, T., Korotkov, A., Bartunov, O.: PostgreSQL 10 documentation: F.5. bloom. https://www.postgresql.org/docs/10/static/bloom.html (2017). Accessed 25 Jan 2018
83. Singh, T., Rangarajan, S., John, D., Henrion, C., Southard, S., McIntyre, H., Novak, A., Kosonocky, S., Jotwani, R., Schaefer, A., Chang, E., Bell, J., Zen, M. Co.: a next-generation high-performance x86 core. ISSCC, pp. 52–53 (2017)
84. Smith, J.E.: A study of branch prediction strategies. In ISCA, pp. 135–148 (1981)
85. Stonebraker, M., Rowe, L.A., Hirohama, M.: The implementation of POSTGRES. TKDE **2**(1), 125–142 (1990)
86. Sun, Y., Hua, Y., Jiang, S., Li, Q., Cao, S., Zuo, P.: SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems. In USENIX ATC, pp. 553–565 (2017)
87. Tarjan, R.E., Yao, A.C.: Storing a sparse table. CACM **22**(11), 606–611 (1979)
88. Tinney, W.F., Walker, J.W.: Direct solutions of sparse network equations by optimally ordered triangular factorization. Proc. IEEE **55**(11), 1801–1809 (1967)
89. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In ICPPW, pp. 207–216 (2010)
90. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L. : Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In ISCA, pp. 191–202 (1996)
91. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In ISCA, pp. 392–403 (1995)
92. Vöcking, B. How asymmetry helps load balancing. In FOCS, pp. 131–141 (1999)
93. Wang, J., Lin, C., Papakonstantinou, Y., Swanson, S.: An experimental study of bitmap compression vs. inverted list compression. In SIGMOD, pp. 993–1008 (2017)
94. Wolfe, M. More iteration space tiling. In SC, pp. 655–664 (1989)
95. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. TODS **31**(1), 1–38 (2006)
96. Yoon, M.: Aging Bloom filter with two active buffers for dynamic sets. TKDE **22**(1), 134–138 (2010)
97. Zhang, H., Lim, H., Leis, V., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: SuRF: practical range query filtering with fast succinct tries. In SIGMOD (2018)
98. Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., Zhang, X.: Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. PVLDB **8**(11), 1226–1237 (2015)