AniFilter: Parallel and Failure-Atomic Cuckoo Filter for Non-Volatile Memories

Hyungjun Oh
Hanyang University, Korea
tcpip15@hanyang.ac.kr

Bongki Cho
Hanyang University, Korea
bongki@hanyang.ac.kr

Changdae Kim
ETRI, Korea
cdkim@etri.ac.kr

Heejin Park
Hanyang University, Korea
hjpark@hanyang.ac.kr

Jiwon Seo*
Hanyang University, Korea
seojiwon@hanyang.ac.kr

Abstract

Approximate Membership Query (AMQ) data structures are widely used in databases, storage systems, and other domains. Recent advances in Non-Volatile Memory (NVM) technologies made possible byte-addressable, high performance persistent memories. This paper presents an optimized persistent AMQ for NVM, called AniFilter (AF). Based on Cuckoo Filter (CF), AF improves insertion throughput on NVM with *Spillable Buckets* and *Lookahead Eviction*, and lookup throughput with *Bucket Primacy*. To analyze the effect of our optimizations, we design a probabilistic model to estimate the performance of CF and AF. For failure atomicity, AF writes minimum amount of logging to maintain its consistency. We evaluated AF and four other AMQs – CF, Morton Filter (MF), Rank-and-Select Quotient Filter (RSQF), and Bloom Filter (BF) – on NVM. We use Intel Optane DC Persistent Memory and Quartz emulation for the evaluation. AF and CF are generally much faster than the other filters for sequential runs. However, in high load factors CF's insertion throughput becomes prohibitively low due to the eviction overhead. With our optimizations, AF's insertion throughput is fastest even in high load factors. In parallel evaluation, AF's performance is substantially higher than CF for both insertion and lookup. Our optimizations reduce the bandwidth consumption, making AF's parallel performance much faster than CF's on bandwidth-limited NVM. For parallel insertion AF is up to 10.7× faster than CF (2.6× faster on average) and for parallel lookup AF is up to 1.2× faster (1.1× faster on average).

*Corresponding author and principal investigator

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387528

1 Introduction

Approximate Membership Query (AMQ) data structures compactly represent a set of items to serve membership tests. AMQs are used in many areas including databases [8, 25, 26], storage systems [10, 22], and network managements [6, 33] for storing items in DRAM, SSDs, and Hard Disks. There have been a large amount of studies to optimize AMQs' performance on those storage media [4, 9, 14, 16, 29, 30].

The advance of Non-Volatile Memory (NVM) technologies such as 3D Xpoint [2] and ReRAM [37] made available byte-addressable persistent memories. These persistent memories have access latencies close to DRAM and data persistency of SSDs. Because of these characteristics – byte-addressability, high performance, and durability – persistent memories provide opportunities and challenges for system software. Much work has been done for database and storage systems to take advantage of the durability and high performance of persistent memories [3, 12, 15, 21, 38]. Especially, persistent index data structures are extensively studied [11, 13, 23, 27, 34, 39–41]. AMQs, as they are often used for index structures, may also take advantage of the persistent memories. Because AMQs often need to be persistent [10, 22, 36], they can exploit the high performance of NVMs. However, existing work on AMQs only study their performance on DRAM, SSD, or Hard Disk, and their performance on NVM is not previously investigated [9, 14, 16, 29].

In this paper we study a persistent AMQ and its optimizations for NVM. We focus on Cuckoo Filter (CF), because it is a good candidate for a persistent AMQ; not considering the evictions, the insertions in CF can be largely handled in a single 8-byte write, which is ideal for failure-atomic persistent data structures. Thus CF performs better than other AMQs on NVM in low load factors. However, in high load factors its performance rapidly declines due to the eviction overhead, making it unusable on NVMs. Based on CF, we present a novel variant called AniFilter (AF)[1]. With our optimizations, AniFilter's performance on NVM is improved by up to 10.7× faster for insertions and 1.2× faster for lookups. This paper contributes the design, analysis, and evaluation of AniFilter. The specific contributions are as follows.

[1]Anis are in the cuckoo family known for having communal nests.

An efficient and persistent AMQ for NVM. We propose three optimizations, namely *Spillable Buckets*, *Lookahead Eviction*, and *Bucket Primacy*. The optimizations are simple to implement and when applied together they collectively achieve high performance on NVM. AF applies logging for failure atomicity; by exploiting NVM's 8-byte failure atomicity unit, AF requires zero or one (and rarely two) additional atomic writes for logging for insertion. For comparison, Rank-and-Select Quotient Filter, an SSD-optimized AMQ, needs more than three cacheline flushes for logging for insertion.

Theoretic analysis of eviction overhead. We present a probabilistic model to predict the eviction overhead of CF and AF. Our model computes the probabilities of slot vacancies, taking evictions into account. The approach in our model is general and thus can be applied to the analysis of other CF variants. The model also calculates the effect of Spillable Buckets, considering *spill-in* and *spill-out* keys and their probabilities. We experimentally verify that our model accurately estimates the eviction probability for CF and AF.

Evaluation of AniFilter and other AMQs on NVM. We evaluated AF and four other AMQs on NVM – CF, Morton Filter (MF), Rank-and-Select Quotient Filter (RSQF), and Bloom Filter (BF); these filters include all state-of-the-art AMQs optimized for DRAM and SSD. We used both a real hardware (Intel Optane DC Persistent Memory) and Quartz emulation to evaluate the sequential and parallel performance. The sequential performance of AF and CF is much better than that of MF, RSQF, and BF. CF's insertion throughput rapidly degrades in high load factors due to the eviction overhead, but the optimizations in AF reduces the evictions making AF fastest for insertion in all load factors. In parallel evaluation, AF's performance is much faster than CF for both insertion and lookup. Our optimizations reduce the eviction overhead and logging overhead for insertion; the optimizations also reduce memory bandwidth consumption for lookup, thereby improving AF's parallel performance.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces AF and our optimizations. Section 4 presents our probabilistic model for the eviction analysis. Section 5 describes AF's failure atomicity and Section 6 describes our parallel implementation. Section 7 evaluates AF and other AMQs. Section 8 discusses the design lessons we learned and Section 9 concludes.

2 Background and Related Work

This section describes the background work to understand our contribution. We overview AMQ data structures and discuss the consistency issues in persistent data structures.

2.1 AMQ Data Structures

AMQ is a set data structure that approximately stores membership information. AMQs commonly support *insert* and *query* operations to add items and to query the membership of an item. Some AMQs support deletion. The query for an item x must return true if x has been added; the query returns false otherwise. With a small probability, however, the query may return true even if x was not previously added.

Bloom Filter (BF) is one of most well-known AMQs [5]. It keeps a zero-initialized bit vector; for the insertion, BF applies k hashes and sets the corresponding bits in the vector. For the query operation, it applies the k hashes and tests the corresponding bits; if all the bits are set, the query returns true, otherwise false. BF has many variants such as Counting Bloom Filter [7] which allows deletion of items, and BloomFlash, which is an SSD-optimized variant [14].

Other AMQs store fingerprints, or hash values, of items instead of setting individual bits [4, 9, 16, 29]. Cuckoo Filter (CF) keeps an array of buckets, each of which contains a fixed number of slots for storing fingerprints [16]. Like Cuckoo Hash Table [28], it applies a hash function to assign an item to a bucket; if all slots in the bucket are full, one slot is selected and its fingerprint is evicted. The evicted one is stored in its alternate bucket which is determined by the other hash function. In high load factors, an insertion in CF may yields a number of evictions, which incurs high overhead on NVMs.

Recently, Breslow and Jayasena proposed Morton Filter (MF) [9], a DRAM-optimized CF variant. MF groups a number of buckets and store them in a *block*. Buckets in a block have a dynamic number of slots, which helps reduce the number of evictions in high load factors. The number of slots for each bucket is written in the header of a block, and the filter needs to sum up these numbers to locate the slots for a bucket. Due to this summation overhead, MF applies batch processing for insertion and query operations; without the batch processing, its performance is lower than CF.

Quotient Filter (QF) is another fingerprint-based AMQ [4]. QF keeps an array of slots. To insert a key, it hashes the key to a fingerprint; the high bits of the fingerprint selects a slot, and the low bits are stored in the slot. If a collision occurs, a variant of linear probing is applied to identify the virtual bucket and locate the fingerprint in the bucket. Because QF stores colliding fingerprints in the ascending order, an insertion may require shifting a number of fingerprints. Pandey et al. recently proposed an SSD-optimized variant, called Rank-and-Select Quotient Filter (RSQF) [29]. RSQF exploits rank-and-select vector instructions to efficiently find virtual buckets [17]. Moreover, a number of slots are grouped into a block for better cache locality. Although it performs well on SSDs, RSQF's performance on NVMs is rather poor as shown in our evaluation because it requires multiple 8-byte updates per insertion for shifting the colliding fingerprints.

2.2 Data Consistency in NVM

NVM data structures must ensure that the stored data is consistent and recoverable in case of a failure. To guarantee consistency and recoverability with volatile CPU caches, changes to the data must be carefully ordered and persisted

in NVM with hardware instructions. Memory fence instructions (MFENCE), such as *mfence* are used for ordering between writes; cache line flushes (CLFLUSH) such as *clflush* are applied to flush data in caches to NVMs. These MFENCE and CLFLUSH instructions are known to be expensive.

Another factor that must be considered for consistency is NVM's atomic write size. In Intel Optane DC Persistent Memory, the first commercially available NVM, the unit of failure atomic write is 8 byte; in other NVMs, the atomic write unit is also expected to be 8 byte [15, 35, 39]. Writes larger than 8 bytes or spanning across 8 byte boundaries are written in multiple atomic writes. In those cases, techniques such as logging or copy-on-write must be applied.

Recently a number of studies have been conducted on persistent data structures for NVMs. Especially indexing structures are extensively studied because they need efficiency and persistency. Many of the proposed persistent indexes are B-tree variants [11, 27, 34, 39]. These B-tree variants apply copy-on-write or append-only write to provide consistency with a reduced number of MFENCE and CLFLUSH.

Hash-based persistent indices are also proposed [13, 40, 41]. PFHT (PCM-Friendly Hash Table) is a variant of Cuckoo Hash Table [13]; it keeps a small supplementary table to reduce the number of evictions for insertion. Level hashing is a write-optimized hash table for NVM [41]. It keeps a sharing-based two-level hash table structure, which helps reducing the number of random accesses per insertion. Although a number of persistent data structures have been studied, our work is the first to demonstrate an efficient AMQ data structure that is specifically targeted for NVMs.

3 NVM-Optimized Cuckoo Filter

This section first gives an overview of Cuckoo Filter. Then it presents AniFilter, our persistent Cuckoo Filter for NVM. We describe the optimizations in AniFilter here; failure-atomic aspect of AniFilter is discussed later in Section 5.

3.1 Cuckoo Filter Overview

CF is structured as an array of buckets, each of which has d slots for storing up to d fingerprints (FP for short). Typical CF implementations keep 4 slots in each bucket [16]. To store the fingerprint FP_x of x, CF applies two hash functions, H_1 and H_2, to determine the bucket index to store FP_x. If any of the two buckets pointed by H_1 or H_2 has an empty slot, FP_x is stored in the slot. Otherwise, one of the slots in the two buckets is selected to evict its fingerprint (FP_e) to store FP_x instead. CF now attempts to store FP_e in its alternate bucket. If the bucket has an empty slot, FP_e is stored there. Otherwise, a slot in the bucket is randomly selected and the FP in the slot is evicted. The process of the eviction and insertion is repeated until an evicted fingerprint can be stored in its alternate bucket without any further eviction. This repeated eviction causes many random access in high load factors, which incurs high performance overhead; the overhead is especially problematic for NVMs as shown in our evaluation because NVM devices have higher latency and lower bandwidth than DRAMs.

Algorithm 1: Inserting a fingerprint into a bucket

```
Input  : AF: AniFilter, i: bucket index, f: fingerprint
Output : Insertion status: {Exist, Inserted, Failed}
1  Function BUCKET_INSERT(AF, i, f)
2      b = AF[i]
3      if f in b ∧ f not in spilled slot then return Exist
4      if ¬full(b) then
5          b[highest empty slot] = f
6          if full(b) then Rearrange FPs for spill encoding;
7          flush_bucket(b)
8          return Inserted
9      for (d = 1; d ≤ Spill_Dist; d++) do
10         b = AF[i + d]
11         if b[0] = f ∧ b[0] is a spilled slot then return
             Exist;
12         else if b[0] = ∅ then
13             b[0] = f
14             if full(b) then Rearrange FPs for spill
                 encoding;
15             flush_bucket(b)
16             return Inserted
17     return Failed
```

Because CF stores FPs and not keys, H_1 and H_2 derive an alternate bucket index with an FP and its current bucket index. A typical hashing scheme is to use $H_1 = hash(x)$ and $H_2 = H_1(x) \oplus hash(x\text{'s FP})$, where $\oplus$ is xor. H_1 can be also computed with H_2 and the FP. To query for key x, $H_1(x)$ and $H_2(x)$ are computed to find the buckets. If x's FP matches any FPs in the two buckets, it returns true, otherwise false.

3.2 Optimizations in AniFilter

Based on CF, we propose AniFilter, a persistent AMQ optimized for NVMs. We implement three optimizations in AF. They are *Spillable Buckets*, *Lookahead Evictions*, and *Bucket Primacy*. The first two improve the insertion throughput by reducing the number of evictions in high load factors, thereby improving the IO performance. The third one, Bucket Primacy, improves query throughput by enabling a subset of queries to access only a single bucket instead of two in CF.

3.2.1 Spillable Buckets

AF allows its buckets to borrow the slots from their neighbor buckets. That is, when inserting an FP into i'th bucket ($b[i]$), if $b[i]$ is full, AF examines the next buckets ($b[i+1]$, ...) for vacancy; if $b[i+k]$ has a vacancy, then the FP is stored in $b[i+k]$, instead of evicting an FP from $b[i]$. We call storing an FP in a borrowed slot in this way as *spilling* an FP. The bucket $b[i+k]$ is called a *spilled* bucket, and the slot in $b[i+k]$

storing the spilled FP as a *spilled* slot. To spill an FP, we check maximum d buckets, where d is called *spill distance*. The default value of d is two, which is decided based on theoretical and experimental analysis (see Section 4). Spilling an FP is likely to be a cache hit, thus is faster than evicting an FP which almost always incurs a cache miss.

Algorithm 1 describes insertion of an FP into a designated bucket with spilling. When spilling an FP in a neighboring bucket, we only allow a designated slot to store a spilled FP, which is the first slot with index zero ($slot_0$, denoted as $b[0]$ in Algorithm 1). The first slot is likely to be in the same cacheline as the designated bucket, hence spilling there is most likely to be a cache hit. The other slots in the bucket encode the information if $slot_0$ stores a spilled FP. The encoding is simple if the bucket is not full; if $slot_1$ is vacant and $slot_0$ stores an FP, then the FP is a spilled one. When normally storing an FP in a partially occupied bucket without spilling, the slots in the bucket are filled in the reverse order of their indices. Thus if a bucket is not full and its $slot_0$ is filled, the slot is guaranteed to contain a spilled FP. If a bucket is fully occupied, the order of the two FPs in $slot_1$ and $slot_2$ encodes the spill information; i.e., if (and only if) the FP in $slot_1$ is greater than the one in $slot_2$, then $slot_0$ stores a spill. The encoding scheme is applied in Algorithm 1 to store an FP in lines 5–6 and to spill an FP in lines 13–14.

Figure 1 demonstrates inserting x (FP_x) and y (FP_y) into AniFilter. The numbers on the left specify the index of each bucket. When storing FP_x in a full bucket $b[1]$ (step ①), we spill the FP in bucket $b[2]$ (step ②); because the bucket is not full, we do not need to rearrange the other FPs in $b[2]$. The steps to store FP_y is described later in Section 3.2.3, but when spilling the FP in bucket $b[5]$, because the bucket is full we swap the FPs in $slot_1$ and $slot_2$ (step ❹) to encode the fact that the first slot stores a spilled fingerprint.

If an FP cannot be stored in the primary bucket referred by H_1 or spilled in its neighbor buckets, the insertion fails. Then we attempt to insert the FP into its alternate bucket referred by H_2; if the insertion still fails, we evict an FP. This process is described in Algorithm 2. We only evict the FPs in its own buckets. Spilled FPs cannot be evicted, because their original buckets are unknown. Lookup runs in a similar way. We first examine the designated buckets. If we cannot find a match, we examine the spillable slots in the next d buckets.

Note that this optimization may slightly increase the false positive rate, because a spilled FP can match for the queries examining any of its d previous buckets. However, for a small value of d, the increment of the false positive rate is negligible. Moreover, Bucket Primacy which is discussed in Section 3.2.3 improves, or decreases, the false positive rate by reducing the number of necessary FP comparisons. Hence the overall false positive rate is improved in AF; we experimentally demonstrate this in the evaluation section.

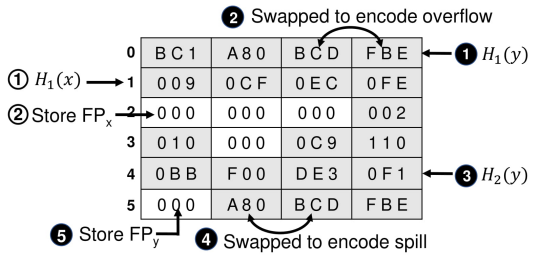


Figure 1. Insertion of x and y with spilling. The slots colored in gray are filled and the slots in white are empty.

3.2.2 Lookahead Eviction

In high load factors, insertions in CF suffer from a large number of evictions [16]. To mitigate this problem, we propose *Lookahead Eviction*, which reduces evictions by carefully selecting the fingerprint to evict.

In Lookahead Evicition, we store each bucket's occupancy information in a bit array called *occupancy flags*. An entry in the array is set if the corresponding bucket is full. The occupancy flags are referred when an insertion causes an eviction in a bucket. To select an FP to evict, we enumerate the FPs in the bucket and evict the one whose alternate bucket is not full according to the occupancy flags. If all the alternate buckets are full, we randomly select an FP to evict. Because this scheme selects and evicts the FP that incurs no further evictions, it helps to minimize the eviction overhead.

To quickly access the occupancy information, we store the occupancy flags in DRAM. Because the size of the occupancy flags is small, its storage overhead is not too significant. If this overhead is non-trivial, we can represent the occupancy information of multiple buckets in a single bit. That is, we can group two consecutive buckets and merge their occupancy information into a single bit; if (and only if) both of the two buckets are full, the corresponding flag is set to one. This reduces the occupancy flags to half, at the cost of performing more random evictions. Because the occupancy flags is a cache, it needs not be persisted. In case of a system failure, we rebuild the flags with the persisted data in the buckets.

3.2.3 Bucket Primacy

In AF, a key x has its *primary bucket*, which is bucket $b[H_1(x)]$ designated by H_1. The alternate bucket $b[H_2(x)]$ is referred as a secondary or alternate bucket. AF always attempts to first insert a key into its primary bucket; its secondary bucket is accessed only if the primary bucket is full. The query runs in a similar way – the secondary bucket is examined only if the primary bucket has no matching FPs.

To reduce the need to access the secondary buckets and the number of random accesses, AF keeps track of each bucket's eviction history – that is, whether a bucket has ever evicted its FP. If a key's primary bucket never performed an eviction before, then the query of the key can be answered

Algorithm 2: Inserting a key into AniFilter

```
Input  : AF: AniFilter, x: key
Output : Insertion status: {EXIST, INSERTED, FAILED}
1  Function INSERT(AF, x)
2      i = H_1(x), f = fingerprint(x)
3      status = BUCKET_INSERT(AF, i, f)
4      if status ≠ FAILED then return status
5      if ¬evicted(AF, i) then
6          Rearrange FPs in AF[i] to mark as evicted
7          flush_bucket(AF[i])
8      for (try = 0; try ≤ MAX_TRY; try++) do
9          i = H_2(i, f)
10         status = BUCKET_INSERT(AF, i, f)
11         if status ≠ FAILED then return status
12         f = evict_slot(AF, i, f)
13         Rearrange FPs in AF[i] to mark as evicted
14         flush_bucket(AF[i])
15     return FAILED
```

by examining only its primary bucket. To this end, we keep the eviction history of each bucket encoded in the order of FPs stored in the bucket. Specifically, the FPs in the last two slots are used to determine whether there was an eviction in the bucket. If a bucket has four slots, the FPs in the third and the forth slots are compared. If the one in the forth slot is larger, the bucket has never previously performed an eviction; then a query to the bucket needs not look up the alternate bucket. In Algorithm 2, line 6 and line 13 denote that the eviction history is updated because an eviction occurred in the bucket.

In Figure 1 when attempting to store FP_y in $b[0]$ (step ❶), the bucket is full and the spillable slots in the following two buckets are also full. Because we cannot store the FP in its primary bucket, the FP should be stored in the alternate bucket. Thus we encode the overflow status in $b[0]$ (step ❷) and try to store it in the alternate bucket $b[4]$ (step ❸). We then spill the FP in $b[5]$ and encode the spill information (step ❹ and ❺).

4 Modeling and Analysis

We design Spillable Buckets and Lookahead Eviction to improve insertion throughput. Lookahead Eviction intuitively improves the throughput by reducing the evictions. For Spillable Buckets, however, it is unclear if it would reduce the evictions altogether. If we spill a key from bucket $b[l]$ to $b[l+1]$, it reduces the evictions for the keys for $b[l]$; but for the keys for $b[l+1]$, the number of evictions may increase.

In this section, we give a theoretic analysis of the eviction overhead for Cuckoo Filter and AniFilter. The analysis for Cuckoo Filter is general and is applicable for the analysis of other Cuckoo Filter variants. Our model estimates the eviction probability for insertion, with and without Spillable Buckets, thereby explaining the performance improvement brought by the optimization. We first introduce some notations. Let $B \geq 1$ and $S \geq 1$ denote the number of total buckets and the number of slots in a bucket, respectively. A bucket is either *full*, *non-full*, or *overcrowded*. A full bucket has exactly S keys; a non-full bucket has less than S keys. An overcrowded bucket has more than S keys. Note that overcrowded buckets are for the analysis purpose only. If an overcrowded bucket has $S+k$ keys, we say the bucket has $k \geq 1$ *excessive* keys. The load factor, or the fullness, of the filter is denoted as $0 \leq \alpha \leq 1$. Then, the total number of keys is αSB. Table 1 shows more notations in our analysis – we also describe them as they are used in the analysis. We first consider the case for CF without Spillable Buckets applied.

Table 1. Notations and definitions for analysis.

Symbols	Description
α	Load factor ($0 \leq \alpha \leq 1$)
B, S	Number of total buckets and slots per bucket
λ_r	Avg. number of keys redistributed at round r
X_r	Avg. number of excessive keys after round r
$A_{k,ij,r}, B_{k,ij,r}$ $C_{k,ij,r}$	Event that a bucket is of type ij[1] and has k non-spilled keys after step 1 (resp. 2 and 3) of round r
$F_{ij,r}, NF_{ij,r}$	Event that a bucket is of type ij[1] and full (resp. non-full) after round r

1. ij is always 00 for Cuckoo Filter thus omitted for Cuckoo Filter analysis.

4.1 Analysis for Cuckoo Filter

Our analysis estimates the eviction probability at a given load factor α, i.e., the probability that an eviction occurs when a key is inserted. The analysis runs in rounds as follows.

In round 0, all αSB keys are distributed to all B buckets by Poisson distribution.

In round $r \geq 1$, the excessive keys in overcrowded buckets are collected and redistributed to non-full buckets by Poisson distribution. After the redistribution, all the overcrowded buckets become full and some of non-full buckets become full or overcrowded.

The number of excessive keys converges to 0 as rounds proceed. In round r, the Poisson distribution parameter, or the average number of keys redistributed to each non-full bucket, is denoted as λ_r. Then the probability $P(\lambda_r, k)$ that k keys are hashed into a bucket is $\lambda_r^k e^{-\lambda_r}/k!$. Let $A_{k,r}$ denote the event that a bucket has k keys after round r. Then, the probability $P(A_{k,r})$ is computed as follows.

$$P(A_{k,r}) = \begin{cases} P(\lambda_0, k)\,, & r = 0. \\ \sum_{i=0}^{\min(k,S-1)} P(A_{i,r-1})P(\lambda_r, k-i), & r \geq 1. \end{cases}$$

At round 0, λ_0 is simply αS. At round $r \geq 1$, λ_r is calculated in the following way. Let F_r (resp. NF_r) denote the event that a bucket is full (resp. non-full) after round r. The probabilities $P(F_r)$ and $P(NF_r)$ are computed as follows.

$$P(F_r) = P(A_{S,r}),\ P(NF_r) = \textstyle\sum_{i=0}^{S-1} P(A_{i,r})$$

Let X_r denote the number of excessive keys in a bucket after round r. Its expected value $E(X_r)$ is computed as follows.

$$E(X_r) = \sum_{i=S+1}^{\alpha SB}(i-S)P(A_{i,r}),\ r \geq 0.$$

The expected number of total excessive keys in all the buckets after round r is $E(X_r)B$. In addition, the expected number of total non-full buckets after round r is $P(NF_r)B$. Thus at round $r \geq 1$, λ_r, or the average number of keys redistributed to each non-full bucket, is computed as follows.

$$\lambda_r = \frac{E(X_{r-1})B}{P(NF_{r-1})B}S = \frac{E(X_{r-1})}{P(NF_{r-1})}S,\ r \geq 1.$$

Because $\alpha \leq 1$, the number of excessive keys converges to 0 as round proceeds. Hence we have the following theorem.

Theorem 1. *The eviction probability for a single insertion in Cuckoo Filter is* $\lim_{r\to\infty} P(F_r)$.

4.2 Analysis for AniFilter

Now we consider the effect of Spillable Buckets. With this optimization, each bucket may borrow slots from next d buckets; a bucket can lend at most one slot to other buckets. We first consider the case when d, the spill distance, is 1. Similarly as the case for CF, the analysis proceeds in rounds as follows.

In round 0, all αSB keys are distributed to all B buckets by Poisson distribution.

Round $r \geq 1$ consists of three steps.

- **In step 1**, we perform spill. For each overcrowded bucket $b[i]$ for $1 \leq i \leq B-1$, the probability that a key is spilled to its next bucket $b[i+1]$ is computed. With the computed probability a single key in $b[i]$ is spilled to $b[i+1]$; with the complementary probability no keys are spilled.
- **In step 2**, the excessive keys are collected from all the overcrowded buckets.
- **In step 3**, The excessive keys are redistributed by Poisson distribution to all non-full buckets and also full buckets from which no key has been spilled out before.

Based on the spill status, each bucket is categorized into four types: 00, 10, 01, and 11. The first bit denotes whether a key was spilled into the bucket (spill-in); the second bit denotes whether the bucket previously spilled a key (spill-out).

Definition 4.1. Let $A_{k,ij,r}$ (resp. $B_{k,ij,r}$ and $C_{k,ij,r}$) for $k \geq 0$, $r \geq 1$, and $i, j \in \{0,1\}$ denote the event that a bucket is ij type and has k non-spilled (its own) keys after step 1 (resp. step 2 and step 3) of round r. For convenience, $C_{k,00,0}$ denotes the event that a bucket has k non-spilled keys after round 0. Let $NF_{ij,r}$ (resp. $F_{ij,r}$) denote the event that a bucket is ij type and non-full (resp. full) after step 3 of round r.

The derivations of $P(A_{k,ij,r})$, $P(B_{k,ij,r})$, and $P(C_{k,ij,r})$ are described in the appendix of this paper. With these probabilities, we can compute the eviction probability as follows.

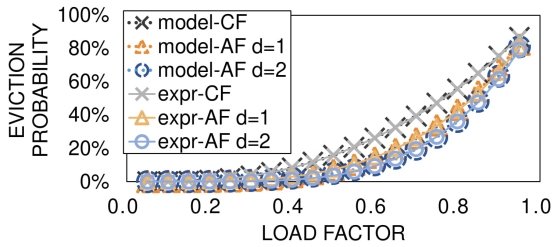


Figure 2. Eviction probabilities by our analysis (dotted lines) and from the experiments (solid lines).

Theorem 2. *The eviction probability for a single insertion in AniFilter with Spillable Buckets, where spill distance is 1, is* $1 - \lim_{r\to\infty}\sum_{i\in\{0,1\}}(P(NF_{i0,r}) + P(F_{i0,r})P(NF_{00,r}|F_{i0,r}))$, *where*

$$P(NF_{i0,r}) = \sum_{k=0}^{S-1-i} P(C_{k,i0,r}),\ P(F_{i0,r}) = P(C_{S-i,i0,r}),\ \textit{and}$$

$$P(NF_{00,r}|F_{i0,r}) = \frac{\sum_{k=0}^{S-1}P(C_{k,00,r})\sum_{k=0}^{\infty}P(C_{k,i0,r})}{(\sum_{k=0}^{\infty}P(C_{k,00,r})+\sum_{k=0}^{\infty}P(C_{k,01,r}))\sum_{k=0}^{S}P(C_{k,i0,r})}$$

Proof Sketch. The eviction probability of a bucket $b[l]$ is computed from its complementary probability that $b[l]$ is either non-full ($P(NF_{i0})$), or full ($P(F_{i0,r})$) and $b[l+1]$ is spillable ($P(NF_{00,r}|F_{i0,r})$). □

For spill distance 2, we need to add two more bucket types (02 and 12). However, this makes the analysis too complicated, so we use a simple approximate analysis. Our approximate analysis first runs the same analysis for spill distance 1. With the converged probabilities of $NF_{00,r}$, $NF_{10,r}$, and $F_{ij,r}$, we compute the probability that an insertion into bucket $b[l]$ incurs an eviction with spill distance 2; here we consider $b[l+1]$ and $b[l+2]$ for spilling. In short, we run the same analysis for spill distance 1, but the final eviction probability is computed considering the spill distance of 2 as follows.

Theorem 3. *The eviction probability for a single insertion in AniFilter with Spillable Buckets, where spill distance is 2, is*

$$1 - \lim_{r\to\infty}\sum_{i\in\{0,1\}}(P(NF_{i0,r}) + P(F_{i0,r})[P(NF_{00,r}|F_{i0,r}) + (1 - P(NF_{00,r}|F_{i0,r}))P(NF_{00,r})])$$

Proof Sketch. The eviction probability of a bucket $b[l]$ is computed from its complementary probability that $b[l]$ is either non-full, full and $b[l+1]$ is spillable, or full and $b[l+1]$ is not spillable and $b[l+2]$ is spillable. □

4.3 Analysis Accuracy

To verify that our analysis accurately estimates the eviction probability, we run the analysis for CF and AF with spill distance 1 and 2 at each 5% load factor. We implemented the code for the analysis and run it for ten rounds, which is enough for the eviction probabilities to converge. We also experimentally measured the eviction probabilities with the actual implementations of CF and AF (with Spillable Buckets only). As we insert random integers into the filters, at 5% load factors we computed the fraction of buckets that would

incur an eviction if a key is inserted to the bucket. Figure 2 shows the results of the analysis and the experiments. In all load factors the estimated values closely follow the observed ones with less than 2% point error. Clearly our analysis gives accurate estimations of the eviction probabilities.

The figure shows that as we increase the spill distance, the gain in the eviction overhead decreases. Further increasing the spill distance to 3 or 4 (not shown in the figure) decreases the eviction probability only by 1–2 % for a fraction of the load factors. Thus AF sets the spill distance 2 by default.

5 Failure Recovery

5.1 Bucket Alignment Issue

In CF, the false positive rate and eviction overhead is determined by its fingerprint size and slot count per bucket [16]. If the fingerprint size is small or the slot count is large, its false positive rate is high. However, if the slot count is small, the eviction overhead becomes high. Thus the trade-off for these parameters is important for the efficiency and performance.

The typical setting is 12 bit fingerprints and 4 slots per bucket [16, 31]. This gives 0.2% false positive rate with good insertion performance in DRAM. Note that decreasing the fingerprint size by a single bit, or increasing the slot count by twice, doubles the false positive rate. Applying this setting for NVM is not straightforward because of the failure atomicity issue. The atomicity unit of NVMs is 8 byte for Intel Optane DC Persistent Memory; it is expected to be the same for other NVMs [15, 35, 39]. With 12 bit fingerprints and 4 slots per bucket, a bucket is 6 bytes in size. Hence updating a bucket may incur two atomic writes, which may cause inconsistency with an incomplete bucket update if a failure occurs.

We solve this data inconsistency issue with logging. Simply setting the fingerprint size to 8 or 16 bits is insufficient. With 8 bit fingerprint and 4 slots per bucket, the false positive rate is 3%, which is too high for many applications. For example, in LSM trees, this setting may incur 10× more unnecessary IO compared to the 12 bit setting. With 16 bit fingerprints, we need 33% more memory space compared to the 12 bit case, which also largely limits its applicability.

5.2 Logging Mechanism

AF uses logging to keep its internal consistency. The size of our log records is 8 byte and we mostly write zero or one log record (two in rare cases) for each FP insertion.

CF typically uses 12 bit fingerprints and 4 slots per bucket to achieve a good false positive rate and performance [16, 31]. In this setting, some buckets are not aligned to 8 byte boundaries and writing to those buckets requires two separate atomic writes. For efficient logging, we group buckets into three types based on their relative locations to 8 byte boundaries. Buckets whose index is (0, 3) mod 4 are Type-A buckets; these buckets are atomically updated, not requiring logging unless eviction occurs. Buckets whose index is 1 mod 4 are Type-B buckets. For these buckets the high 16 bits are first written before the low 32 bits are written. Buckets whose index is 2 mod 4 are Type-C buckets. For these buckets the low 16 bits are written before the high 32 bits are written. We always update the smaller part of the buckets (16 bits) first to record minimal amount of logs. Our scheme exploits the memory model of recent Intel processor families, specifically the fact that in the processor caches, the writes to a same cacheline are never reordered when they are persisted to NVM [1, 15]; for a small subset of buckets that are stored across cacheline boundaries, we enforce the ordering of the two writes for the bucket updates.

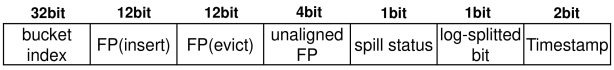


Figure 3. Log record format.

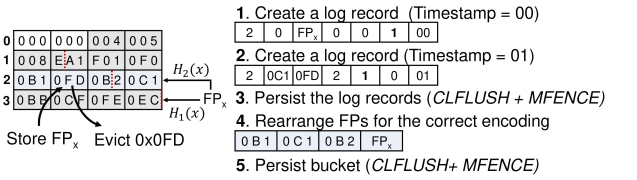


Figure 4. Example logging case for a Type-C bucket.

Figure 4 shows an example logging case. The details of the example will be described later in this section, but on the left is the buckets and their alignments to 8 byte boundaries (the red dotted lines). On the left side of each bucket are their indices. Bucket 0 and 3 are Type-A (aligned to 8 byte boundaries), bucket 1 is Type-B, and bucket 2 is Type-C.

AF keeps two most recent logging records in NVM for recovery. Each logging record is 8 byte in size, thus in total we use 16 bytes for two log entries. A new log record is written in one of the two log entries in a round robin manner. Figure 3 shows the format of our log records. It shows the descriptions of the fields and their bit widths, which add up to 64 bits or 8 bytes. If an insertion of fingerprint FP_x to bucket $b[i]$ causes an eviction, the bucket index i (32 bits), the inserted fingerprint FP_x (12 bits), and the evicted fingerprint FP_y (12 bits) are first written in the log record. For Type-B and Type-C buckets, the FP in $slot_1$ and $slot_2$ – i.e., the slots that are stored across an alignment boundary – are written in the log record; we only store the 4 bits of the unaligned FP, which is sufficient for the recovery (see Section 5). The next bit denotes the spill status of the bucket – it is set if $slot_0$ stores a spilled FP. The next bit, called a *log-splitted bit*, is set if two log records are needed for a single insertion/eviction; only Type-C buckets need this bit in a rare case. The following two bits, or log timestamp bits, are used to record the order between the two log entries.

When updating the slots in a bucket, we first write a log record (or two in a rare case) if the update incurs an eviction or the update is not atomic. If the update is atomic and does not incur an eviction, we need not record a log entry.

Note that we need two log records only for Type-C buckets (and a in rare case); for all other bucket types, we write zero or one log record. After we *CLFLUSH* the log record, we issue *MFENCE* to make sure that the log record is persisted on NVM before we update the bucket. Depending on the bucket types and the slot index of the evicted FP, the logging proceeds in a slightly different manner. For example, if an eviction occurs in Type-A buckets, the log record only keeps the bucket index, the FP to insert, and the FP to evict. However, for Type-B and Type-C buckets, we need to store more information for recovery, such as 4 bits of the unaligned FPs. In the following section, we use a specific example to demonstrate how the logging works in AF.

5.3 Example Logging Cases

Figure 4 shows an insertion to a Type-C bucket. It requires logging two records because 1) FP_x was not previously evicted thus there is no previous record containing FP_x, and 2) FP_3 is overwritten during the update, which requires logging FP_3. We need two log entries only in this particular case.

As previously explained, when we update this bucket in two separate atomic writes, we first update the (smaller) low 16 bits (4 bits of $slot_2$ and $slot_3$), after which we update the high 32 bits ($slot_0$, $slot_1$, and 8 bits of $slot_2$). Before updating the bucket, we create a log record, write FP_x in the evicted FP field, and set the log-splitted bit in the record; only four fields (bucket index, evicted FP, log-spilitted bit, and timestamp) are used in this record. Then we write the second log record (step 2 in the figure), where we store the fingerprint in $slot_3$ in the field we normally store the fingerprint to insert. We also store in the second record the fingerprint to be evicted, 4 bits of the fingerprint in $slot_2$, and the bucket's spill status. We persist the two log records before we attempt to update the bucket. Because the two log records are stored in a same cacheline, the writes to these records are never reordered [1, 15] and we do not need two separate *CLFLUSH*/*MFENCE*'s. Finally we update and persist the bucket.

5.4 Recovery

If a failure occurs, we check the consistency by examining the two log records in NVM. The recovery proceeds in the following way:

1. From the record, FP to insert (FP_i), FP to evict (FP_e), and bucket index are read. The corresponding bucket B is read.
2. We check if FP_e and FP_i are contained in the bucket. The following four cases are possible:
   (a) $FP_e \in B$ and $FP_i \in B$, (b) $FP_e \notin B$ and $FP_i \notin B$
   (c) $FP_e \in B$ and $FP_i \notin B$, (d) $FP_e \notin B$ and $FP_i \in B$
3. For case (a), (b), and (c), we undo the partial update and re-apply the insertion. For (d), we further examine if the update is incomplete, and proceed the recovery.

The case (a) and (b) indicate that the bucket is partially updated. The case (c) means that either the bucket is partially updated or not updated at all; for the latter case, applying the undo operation makes no changes to the bucket. For (d), we use the information in the log records to determine if the update is complete; we elaborate on this below. For the interest of the space, we do not include a proof for the correctness of the recovery; instead, we give an intuitive explanation with a concrete example.

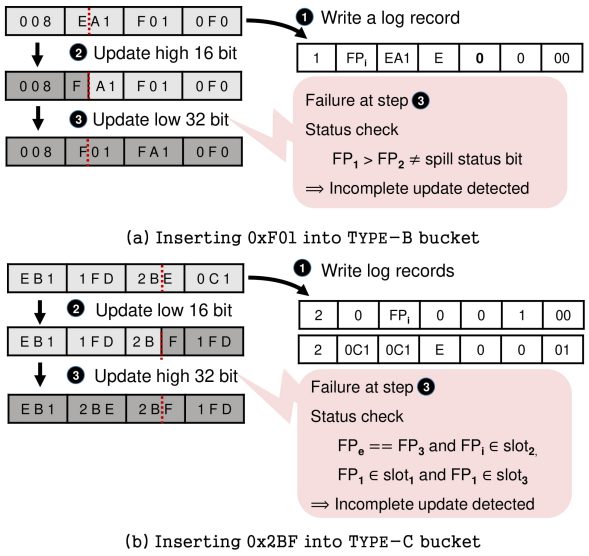


Figure 5. Example recovery for Type-B and Type-C buckets.

Type-B and Type-C buckets have 8 byte alignment boundaries within the bucket in $slot_1$ and $slot_2$ respectively, thus the recovery for those two types is more complicated than for Type-A buckets. Between Type-B and Type-C buckets, the recovery for Type-B is simpler, because we never re-arrange the FP in $slot_0$ for Type-B buckets – it is either evicted or stays in $slot_0$ when the bucket is updated. Hence it is simple to determine if the update is complete when a failure occurs, because for incomplete updates only the first four bits of FP in $slot_1$ is updated. The recovery logic for Type-B is as following. For case (d) of Type-B buckets, we can confirm the completion of the update by examining if the spill encoding in the bucket is correct – by comparing it against the spill information in the log. For incomplete updates for case (d), the spill encoding in the bucket must be incorrect. That is, if updating first half of a Type-B bucket results in case (d), the update is either complete (low 8 bits of FP_i is equivalent to that of FP_e in $slot_1$) or the spill encoding is incorrect (FP_2 or FP_3 should be stored in $slot_1$ but only its first four bits are copied). The latter may occur if the high 4 bits of FP_i equals that of FP_2 or FP_3, and the low 8 bits of FP_i equals that of the FP in $slot_1$. Figure 5 (a) illustrates this case; $FP_i[0..3] = FP_2[0..3] = 0xF$ and $FP_i[4..11] = FP_1[4..11] = 0xA1$. In this example, the spill encoding is incorrect, thus the update is incomplete; we undo the partial update and re-apply the insertion.

For Type-C buckets, the FPs in $slot_1 - slot_3$ can be rearranged during the update; in a partial update, the FP in $slot_3$ can be replaced by any of the FPs in $slot_1$, $slot_2$, or FP_i. This yields many cases to consider, making the recovery complicated. For Type-C buckets, we mainly use three information in the log for the recovery: the FP to insert (FP_i), the FP to evict (FP_e), and the FP in $slot_3$ (FP_3). Let us consider the case (d). Suppose that the update was incomplete. Then FP_e must be either the FP_2 or FP_3; FP_e cannot be the FP in $slot_1$ for incomplete updates, because we always update the latter 16 bits of the bucket first. Let us first consider the case when FP_e is FP_2 (that is, FP_e is different from FP_3 in the log); we test if FP_3 is in the bucket and check if the spill status bit in the log matches. The update is complete if both tests pass, similarly as the case for Type-B bucket described above.

Now we consider the case when FP_e is FP_3. If FP_i is in $slot_0$ or $slot_1$ the update is complete. Two sub-cases that FP_i is in $slot_2$ or $slot_3$ are left. If FP_i is in $slot_3$, the update must be complete because in such case (and under the condition that $FP_e \notin B$ and $FP_i \in B$) the FPs in $slot_1$ and $slot_2$ need not be updated; i.e., the spill status does not change. In the remaining sub-case when FP_i is in $slot_2$, the only way an incomplete update results in case (d) is when FP_1 is repeatedly stored in $slot_1$ and $slot_3$; hence we test this to find out if the update is complete and proceed accordingly. Figure 5 (b) illustrates this last sub-case. A system failure occurred after step 2, at which point FP_i (0x2BF) is in the bucket and FP_e (0x0C1) is not, but the update is incomplete. After we find out that FP_i is in $slot_2$, we test if FP_1 (0x1FD) is repeatedly stored in $slot_3$; because this is the case, we determine that the update is incomplete.

6 Parallel Implementation

We implement parallel AF and CF with fine-grained locking. For the typical setting of 12 bit FPs and 4 slots per bucket, a bucket cannot be updated atomically without reading/writing neighboring buckets. Thus we statically group four consecutive buckets and assign a single shared lock for those buckets, so that access to the buckets is synchronized; i.e., bucket index modular four designates the shared lock for those four buckets. The fine-grained lock is implemented with a bit array and atomic test-and-set instruction.

Similarly as in parallel Cuckoo Hashing [24], we make operations to acquire the locks for both primary and secondary buckets of an FP. For lookup, we know the two buckets in advance thus we acquire the two locks (or one if the buckets share a lock) with lock ordering. For examining spills, we try acquiring the lock for the spill bucket (unless we already hold it) while holding the two locks; if we fail to acquire the lock, we release the two locks and restart to avoid deadlock. If we succeed, we examine the spill slot and release the lock.

For insertion, we also acquire the locks for the two buckets with lock ordering. For spills, we try acquiring the lock and if it fails, we do not attempt to spill. If an eviction is needed, we try acquiring the lock for the other bucket of the evicted FP. If we fail, we select another slot to evict and try acquiring the lock again. If succeed, we evict the selected FP and insert the new FP while holding the three locks. Then we release the lock for the other bucket of the inserted FP. Now with the two locks held for the evicted FP, we continue the remaining insertion process. This way, we always have the locks acquired for the primary and secondary buckets for inserted and/or evicted FPs, preventing any concurrent access to those buckets. Hence any lookup or insertion cannot read inconsistent states. In AF, we further optimize the synchronization by distinguishing new insertions and insertions of evicted keys. For new insertions, AF does not acquire two locks for two buckets at the same time, but acquires a single lock for a primary bucket and then another lock for a secondary bucket. This optimization improves the parallel insertion throughput in low load factors (See Section 7.2).

This synchronization scheme, however, has the risk of livelock, which may occur 1) between an insertion and lookups or 2) between multiple insertions. For the former, an insertion, while holding the lock for bucket $b[l]$, may repeatedly try acquiring a lock for the eviction – a lock for the other bucket of one of the four FPs's in $b[l]$. If four concurrent lookups, each holding a lock for those four buckets, are waiting to acquire the lock for $b[l]$, this results in a livelock. This can be solved by forcing a lookup to have a bounded lock wait time. If a lookup cannot acquire the second lock in a given time, we release the acquired lock and restart the lookup, thereby avoiding the livelock probabilistically.

The latter case, where a livelock occurs between multiple insertions, cannot be solved with the bounded lock waiting and restarting. While inserting an evicted FP, if we release all the locks and re-acquire them, a lookup may interfere and incorrectly reply that the evicted FP is not in the filter. Fortunately though, the probability of the livelock between insertions is extremely low. If the maximum number of concurrent insertions is I and the number of buckets is B, then the livelock probability is no larger than $(2I/B)^4$; for each of the four FPs, the locks for their other buckets must be held by an insertion thread. If $I = 20$ and $B = 2^{28}$, then the probability is 10^{-28}, which is much lower than the memory bit error rate. Because the livelock between multiple insertions is unlikely to occur, we do not attempt to solve the problem.

Furthermore, we modified the logging mechanism in our parallel CF and AF. First we allocate thread-local log entries. Second we explicitly mark the completion of bucket updates in the log records. Without this explicit commit mark, we cannot determine the order of updates to a bucket if multiple threads write to a same bucket; the writes are synchronized, but if a failure occurs we cannot determine the write order from the logs. The commit mark makes it simple to find the log records for on-going updates in case of a failure.

7 Evaluation

7.1 Experiment Setup

We have evaluated the performance of AF on NVM. We used Intel Optane DC Persistent Memory (Optane DC PM for short), because it is the only commercially available NVM at the time of evaluation. The random read and write latency for Optane DC PM are 305 ns and 94 ns respectively [19]. We used *App Direct* mode to directly access NVM without DRAM intervention. We also used Quartz, a DRAM-based NVM latency emulator to examine the performance of AF on other NVMs of different technologies. For the Quartz evaluation, we varied the read latency from 200 to 500 ns; we set the write latency to be between 100 ns and 500 ns. To emulate the write latency, we manually injected delays after *CLFLUSH* instructions [18, 20, 23, 32, 35]. We do not inject stall cycles for each store instruction because the latency of write without memory fence is mostly hidden by CPU cache. We run the experiments on a machine with Intel Xeon Gold 5215M running at 2.5GHz frequency; the CPU has 10 cores and 20 hardware threads. The machine has two CPUs and we used *numactl* command to force all threads to run on a single CPU. The machine has 384 GB DRAM and 1512 GB NVM (Intel Optane DC PM). We used Linux Kernel version 4.18.0 for compiling and running the experiments. The experiments on Quartz emulation are executed on a machine with Xeon CPU E5-2620 running at 2.4GHz frequency and with 96 GB DRAM; Linux Kernel version 4.8.12 is used.

We compare the performance of AF with four other AMQs – Cuckoo Filter, Morton Filter (MF), Rank-and-Select Quotient Filter (RSQF), and Bloom Filter (BF). To the best of our knowledge, these are the state-of-the-art AMQs; MF is optimized for DRAM and RSQF is optimized for SSD. There is no other AMQs specifically optimized for NVM. We do not evaluate other BF variants because BF generally runs much slower than the other filters. For the four AMQs, we used the reference implementations specified in the papers. For Optane DC PM evaluation, we modified the code to allocate the filters on NVM; for the Quartz evaluation, we modified the code to inject stall cycles after *CLFLUSH* instructions for the write latency emulation. We also implemented the logging for the four filters, so that the data consistency is guaranteed. For a fair comparison, we set the theoretical false positive rates (FPR) of the filters to be 1/512. For MF, however, it does not support the configuration for 1/512 FPR, thus we use its default setting (8 bits FP and 3 logical slots per bucket) that gives higher FPR; note that this setting is the same as how MF is evaluated by Breslow et al [9]. Table 2 summarizes the settings of the filters. Because AF optionally uses DRAM for caching bucket occupancy flags, we report two settings for AF with and without the optional DRAM cache. We also test AF with 16 and 8 bit fingerprints, the settings that use simpler logging.

Table 2. Settings of the five filters.

	AF	CF	MF	RSQF	BF
Max Item Count	1.02G	1.02G	1.02G	1.02G	1.02G
Filter Size (GB)	1.50/1.53	1.50	1.39	1.39	1.55
Bits per Element	12.63/12.88	12.63	11.71	11.71	13.04
Theoretical FPR (%)	0.19	0.19	0.32	0.19	0.19
Experimental FPR (%)	0.16	0.19	0.31	0.19	0.19

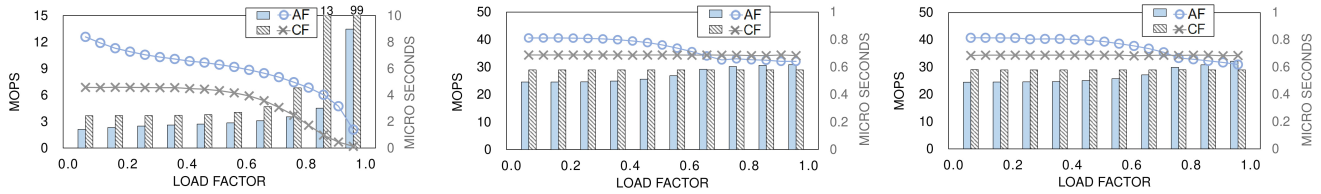
Note that AF's experimental FPR is 15% lower due to Bucket Primacy. When querying for a key in AF, if its primary bucket did not previously evict any FP, we only match with the FPs in the primary bucket. While Spillable Buckets can potentially worsen the FPR, the effect of Bucket Primacy is significantly higher, thus AF shows higher FPR in practice.

We randomly generated 64-bit integers for the input data. By default we used 2^{30} number of integers for the evaluation; similar (or smaller) input size is used for evaluation in previous work [16, 29]. We measure the throughput of the insertion and lookup at every 5% load factor (5, 10, ..., 95%). We first measure the insertion and lookup throughput at the same time. We insert the input data that amounts to 5% of the filter capacity and measure the insertion throughput. Then we measured the negative lookup throughput with uniform random integers, after which we measured the positive lookup throughput; for the lookup, we also used integers that amounts to 5% of the filter capacity. Unless stated otherwise, all the reported results are the average of five runs.

7.2 Parallel Performance

This section evaluates the parallel performance of CF and AF. We exclude MF, RSQF, and BF from this evaluation because they have 1) high synchronization overhead and 2) high memory bandwidth consumption. MF has very low insertion throughput without its batch processing optimization, which cannot be efficiently applied for parallel implementation. Also, MF has high synchronization overhead because it needs to access the meta data of many buckets for inserting a single value. For RSQF, an insertion of a single FP may shift a number of FPs, thus preventing concurrent access to those shifted FPs. BF randomly updates K bits, requiring synchronization for those K locations for each operation. Also RSQF and BF retrieve a number of data for each insertion and lookup, thus their parallel performance cannot scale on NVM for its limited bandwidth. Note that the sequential performance of MF, RSQF, and BF on NVM is already not good (see Section 7.4), and the parallel performance of CF and AF on NVM is bandwidth limited (discussed later in this section). For the parallel evaluation, we run all the experiments with 8 to 32 threads and report the evaluation results with 20 threads because it gives the maximum performance for both CF and AF. We used *numactl* command to make sure that all the threads are running on the same CPU and all the memory access is served by local NVM and DRAM.

Figure 6 shows the parallel throughput (line plots) and latency (bar plots) of CF and AF. The insertion throughput of AF is substantially higher than that of CF in both low and



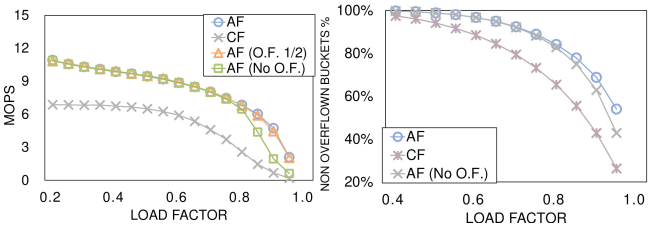
(a) Insertion Throughput **(b)** Successful lookup Throughtput **(c)** Uniform Random Lookup Throughput

Figure 6. Parallel performance of CF and AF with 20 threads. Throughput (line plot) and latency (bar plot) are shown.

high load factors. In low load factors (< 0.4), the speedup primarily comes from 1) our synchronization optimization and 2) our optimized logging based on bucket types. AF distinguishes new insertions from insertions of evicted keys; for new insertions, AF acquires a single lock for a primary bucket and then another for an alternative bucket instead of acquiring two locks at the same time. This optimization and our optimized logging may be readily integrated into CF to improve its insertion throughput. In higher load factors, however, the speedup is attributed to Spillable Buckets and Lookahead Eviction. The two optimizations reduce the number of random access and thus decrease the memory bandwidth consumption. For insertion, AF achieves up to 10.7× higher throughput (at load factor 95%) than CF. On average, AF's insertion throughput is 2.6× higher than CF.

Figure 6 (b) and (c) respectively show the parallel throughput for successful lookup and random lookup (representing unsuccessful lookup). In low load factors, AF's throughput for both successful and unsuccessful lookup is higher thanks to Bucket Primacy; the optimization helps to spare the access to secondary buckets if primary buckets did not overflow. For successful lookup, AF is up to 1.18× faster than CF (1.07× faster on average). For unsuccessful lookup, AF is up to 1.18× faster than CF (1.09× faster on average). In high load factors, the overhead of examining spilled buckets slows down AF, making it run slightly slower (5 – 10%) than CF.

For both insertion and lookup, the throughput is limited by the NVM's memory bandwidth. Here we compute the memory bandwidth consumption of AF in the parallel evaluation. Because AF accesses a single bucket at a time, the access granularity is mostly cacheline size, or 64 bytes in our setting. In low load factors, insertion and lookup almost always access a single bucket; insertion requires logging in half of the time (in low load factors). Based on this, we calculate the memory bandwidth consumption of insertion and lookup. For insertion the bandwidth consumption is computed to be 1.49 GB/s and for lookup it is 2.61 GB/s. These are close to the maximum read and write bandwidth for random access in Optane DC PM; the maximum bandwidths are reported 1.5 GB/s for random writes and 2.8 GB/s for random reads [19].



(a) Insertion throughput **(b)** % of non-overflown buckets

Figure 7. Effect of Lookahead Eviction. 'O.F. 1/2' is occupancy flags reduced to half; 'No O.F.' is no occupancy flags.

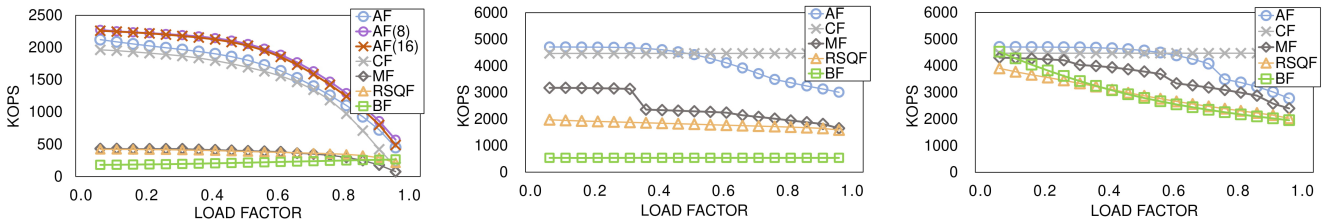
7.3 Effect of Optimizations

We investigated the effect of our optimizations and their synergistic effect. We first look into Lookahead Eviction, which requires additional memory (DRAM). Lookahead Eviction keeps the occupancy flags in DRAM to check whether each bucket is full or not; by reducing the number of evictions in high load factors this optimization improves the insertion throughput. In Figure 7 (a), we plotted AF's insertion throughput with the occupancy flags reduced to half (one per two buckets) and without the occupancy flags. Clearly in load factors higher than 90%, Lookahead Eviction has a large impact on the performance. Reducing the occupancy flags by half does not worsen the impact, thus by using 0.12 extra bit per slot, it triples the performance in 95% load factor.

Furthermore we evaluated synergistic effect between the optimizations; i.e. we measured how Spillable Buckets and Lookahead Eviction impact Bucket Primacy. Figure 7 (b) shows the increase of the percentage of non-overflown buckets when Spillable Buckets and Lookahead Eviction are applied. Non-overflown buckets, or buckets that did not previously evict any FPs, help lookup operations to access only a single bucket instead of two. Spillable Buckets and Lookahead Eviction substantially reduce the number of evictions thereby boosting the effect of Bucket Primacy optimization.

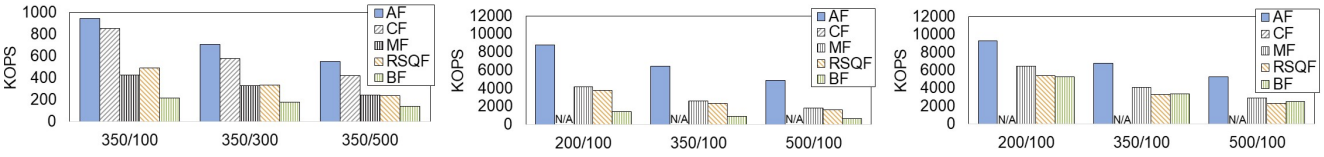
7.4 Sequential Performance

Figure 8 compares the sequential performance of all five filters (AF, CF, MF, RSQF, and BF) on Intel Optane DC PM. The same machine for the parallel evaluation is used. Let us examine the insertion throughput. Both AF and CF show much



(a) Insertion Throughput **(b)** Successful Lookup Throughput **(c)** Uniform Random Lookup Throughput

Figure 8. Sequential performance of the four filters. For insertion, AF with 8 and 16 bit fingerprints are also evaluated.



(a) Insertion Throughput **(b)** Successful Lookup Throughtput **(c)** Uniform Random Lookup Throughput

Figure 9. Performance of the four filters on Quartz emulation (75% load factor). For insertion, read latency is set 350 ns and write latency is varied 100 – 500 ns. For lookup, write latency is set 100 ns and read latency is varied 200 – 500 ns.

higher throughput than MF, RSQF, and BF in low load factors. As in the parallel evaluation, CF's throughput quickly degrades in high load factors due to the eviction overhead, but AF's throughput remains higher than all other filters. Morton Filter is evaluated without its batch optimization – all its other optimizations are applied. When the batch optimization is applied, MF's insertion throughput is improved by 5.5× and its lookup throughput is improved by 2.9× on average; that makes MF's insertion throughput on par with AF's and MF's lookup throughput 2 times faster. However, batch optimization cannot be efficiently integrated into a parallel implementation without incurring high latency, and thus here we show MF's sequential performance without the batch optimization. Note that for failure atomicity, MF must store the entire block's data (for 64 logical buckets in this evaluation) in the log as well as meta data such as the block index and inserted/evicted keys. The necessity of MF's large amount of logging as well as its high synchronization overhead makes its parallel implementation not scale well on bandwidth-limited NVMs. RSQF's insertion throughput is low, because it needs to shift a number of fingerprints, which incurs a large amount of logging. For BF, each insertion updates k random locations, where k is the number of hashes (k=9 in our experiment); this degrades the insertion throughput. Both RSQF and BF access a large amount of data per insertion, thus the two filters are not practical on NVM having a limited memory bandwidth. We also show the performance of AF with 16 and 8 bit fingerprints, that use simpler logging; they perform slightly better due to the reduced logging but not by much. Note that Spillable Buckets and Lookahead Eviction are still effective for those cases.

Figure 8 (b) and (c) respectively show the throughput for successful and unsuccessful lookup for the five filters. For successful lookup, MF's throughput is lower than AF and CF because of the counting overhead to locate a bucket's slot in its block. RSQF and BF show low throughput because they access multiple cachelines per operation. The filters also require more computation than CF and AF; RSQF performs rank-and-select computation, which is expensive. BF computes multiple hash values. For unsuccessful lookup, however, MF, RSQF, and BF perform fast in low load factors, because the filters can examine small amount of data to determine the nonexistence of an item. For example, BF may return false if the bit for any hash is unset. In high load factors, their throughput drops because this kind of short-circuit checking is mostly not applicable.

Now let us examine the lookup throughput of AF and CF. In low load factors, AF is only 10% faster than CF. Although Bucket Primacy enables AF to examine a single bucket for most lookup, CF issues two memory access requests for two buckets at a same time. This makes CF use more bandwidth, but the sequential performance is not bounded by NVM's bandwidth. In high load factors, the throughput of AF decreases as the need to check the secondary buckets increases; there is also the overhead of checking spilled buckets. These overhead makes AF's throughput 33% and 38% lower than CF for successful and unsuccessful lookup for sequential runs.

7.4.1 Effect of Latency Changes

We investigate how the throughput of the five filters changes for varying NVM latencies. We used Quartz emulator for this experiment to control the read and write latency. To examine how insertion throughput changes with different NVM latencies, we vary the write latency from 100 to 500 ns and fixed the read latency to 350 ns. Figure 9 shows the measured throughput at 75% load factor. Clearly AF's insertion throughput is substantially higher than the other filters at all

three write latencies. For the lookup throughput we set the write latency to 100 ns and vary the read latency from 200 ns to 500 ns. Figure 9 (b) and (c) shows the throughput of successful and unsuccessful lookup respectively. The numbers for CF are not available because Quartz does not correctly emulate the NVM latency when two memory requests are issued at a same time – for such case, incorrect longer delay is injected by Quartz. For different read latencies, AF shows much higher throughput than MF, RSQF, and BF.

8 Design Lessons

When designing the optimizations for AniFilter, we mainly investigated the trade-off of memory requests and computation. Spillable Buckets, for example, reduces random memory accesses but requires encoding and decoding spill information. Because NVM has higher latency than DRAM, trade-off of memory requests with more computation has a chance to improve the performance. However, the operations in Cuckoo Filter are very light-weight with a small amount of computation each, and NVM's latency is only 2–3× higher than that of DRAM; any technique that involves heavy computation would be too expensive for AniFilter.

Considering the characteristics of Cuckoo Filter and NVM, we designed the optimizations to be simple and computationally light-weight. For Spillable Buckets and Bucket Primacy, encoding and decoding of spill/overflow status needs just a few comparisons and conditional branches; adding those branches may cause stalls in the instruction pipeline, but with NVM's higher latency this overhead is relatively inexpensive. Because of the branch overhead, these two optimizations are not effective in DRAM with lower latency. On the other hand, Lookahead Eviction is fairly effective even in DRAM, because it requires only simple lookup of occupancy bits. When it is applied in DRAM, Lookahead Eviction improves the insertion throughput by up to 2× in high load factors and 1.15× on average.

For failure atomicity, we decided to use logging, because in many cases bucket updates are done in a single 8-byte atomic write; thus logging is needed less commonly. For other persistent data structures updating a larger size of data for each operation, an alternative approach, such as copy-on-write, needs to be (also) explored. We judiciously designed AniFilter's logging to minimize the size of log record and the number of atomic writes. For updating unaligned buckets (Type-B and Type-C) in two atomic writes, we always update the smaller parts first; this way, we need to log only 4 bits of the fingerprints that are stored across 8-byte alignment boundary for failure recovery, making the log record fit in 8 byte. We presumed that using two (8-byte) log records is minimum and did not further optimize it. Our rationale is that in the worst case for Type-C buckets, we need to record the bucket index (32 bits), fingerprint to insert (12 bits), fingerprint to evict (12 bits), and fingerprint in $slot_3$ which is being overwritten (12 bits); these altogether add up to 68 bits and cannot be packed in a single 8 byte log record.

9 Conclusion

This paper presents AniFilter, an optimized Cuckoo Filter for NVM. Based on Cuckoo Filter, we propose three optimizations – *Spillable Buckets*, *Lookahead Eviction*, and *Bucket Primacy* to improve its throughput on NVM. To analyze the effect of our optimizations, we design a probabilistic model to estimate the eviction overhead of Cuckoo Filter and AniFilter. Our model accurately estimates the eviction probability per insertion with the estimation error smaller than 2% point. For failure atomicity, AniFilter writes minimum amount of logging to maintain its consistency in case of system failure. We evaluated AniFilter and four other AMQs – Cuckoo Filter, Morton Filter (MF), Rank-and-Select Quotient Filter (RSQF), and Bloom Filter – on NVM for sequential and parallel performance. We use both a real hardware (Intel Optane DC Persistent Memory) and Quartz emulation for the evaluation. For sequential execution, AF and CF are generally much faster than MF, RSQF, and BF. However, CF's insertion throughput is prohibitively low in high load factors due to the eviction overhead. With our optimizations AF's insertion throughput is highest in all load factors. In parallel evaluation, AF's performance is substantially higher than CF for both insertion and lookup. Our optimizations significantly reduce the bandwidth consumption, making AF's parallel performance much faster than CF's on bandwidth-limited NVM. For parallel insertion AF is up to 10.7× faster than CF and for parallel lookup AF is up to 1.2× faster.

Acknowledgement

This work is supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A1A02086132) and by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1300). We thank anonymous reviewers for their comments. We thank Beomseok Nam and Sam H. Noh for the discussions. The corresponding author is Jiwon Seo.

References

[1] Intel 64 and ia-32 architectures software developer's manual volume 3a, section 8.2.2. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[2] Intel and Micron Produce Breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of SIGMOD*, pages 707–722. ACM, 2015.

[4] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *In Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[6] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of SIGCOMM*, volume 36, pages 315–326. ACM, 2006.
[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Proceedings of ESA*, pages 684–695. Springer, 2006.
[8] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the VLDB Endowment*, pages 323–333. Morgan Kaufmann Publishers Inc., 1984.
[9] A. D. Breslow and N. S. Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *In Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.
[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *In TOCS*, 26(2):4, 2008.
[11] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *In Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.
[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of SIGOPS*, pages 133–146. ACM, 2009.
[13] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. Revisiting hash table design for phase change memory. *In SIGOPS*, 49(2):18–26, 2016.
[14] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of ICDCS*, pages 635–644. IEEE, 2011.
[15] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of EuroSys*, page 15. ACM, 2014.
[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of CoNEXT*, pages 75–88. ACM, 2014.
[17] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings of WEA*, pages 27–38, 2005.
[18] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *In Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.
[19] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.
[20] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in write-ahead logging. *In Proceedings of ASPLOS*, 50(2):385–398, 2016.
[21] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proceedings of SOSP*, pages 460–477. ACM, 2017.
[22] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *In SIGOPS*, 44(2):35–40, 2010.
[23] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of FAST*, pages 257–270, 2017.
[24] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.
[25] L. F. Mackert, G. M. Lohman, et al. R* optimizer validation and performance evaluation for distributed queries. *Readings in database systems*, pages 219–229, 1988.
[26] J. K. Mullin. Optimal semijoins for distributed database systems. *In TSE*, (5):558–560, 1990.
[27] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of SIGMOD*, pages 371–386. ACM, 2016.
[28] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
[29] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of SIGMOD*, pages 775–787. ACM, 2017.
[30] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.
[31] K. A. Ross. Efficient hash probes on modern processors. In *Proceedings of ICDE*, page 1297–1301. IEEE, 2007.
[32] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh. Failure-atomic slotted paging for persistent memory. *In Proceedings of ASPLOS*, 45(1):91–104, 2017.
[33] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *In SIGCOMM*, 35(4):181–192, 2005.
[34] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of FAST*, volume 11, pages 61–75, 2011.
[35] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of ASPLOS*, volume 39, pages 91–104. ACM, 2011.
[36] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.
[37] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie. Understanding the trade-offs in multi-level cell reram memory design. In *Proceedings of DAC*, pages 1–6. IEEE, 2013.
[38] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of SOSP*, pages 478–496. ACM, 2017.
[39] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of FAST*, volume 15, pages 167–181, 2015.
[40] P. Zuo and Y. Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of MSST*, 2017.
[41] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of OSDI*, pages 461–476, 2018.

Appendices

A Supplementary Lemmas for Section 4.2

Lemma 1. *A key is spilled from bucket $b[i]$ to $b[i+1]$ during step 1 if and only if*
(1) $b[i]$ was overcrowded and of type 00 (resp. 10) before step 1 and it is of type 01 (resp. 11) after step 1, and
(2) $b[i+1]$ was non-full and of type 00 before step 1 and it is of type 10 after step 1.

Definition A.1. Let $I_{k,ij,r}$ (resp. $O_{k,ij,r}$) for $k \geq 0, r \geq 0$, and $i, j \in \{0, 1\}$ denotes the event that a bucket has k non-spilled keys and it is of $i\ j$ type before the step 1 of round r, and a key is spilled into (resp. out from) the bucket during the step 1 of round r.

Lemma 2. *At the end of step 1 of round r,*

$$P(A_{k,00,r}) = \begin{cases} (1)\ P(C_{k,00,r-1}) - P(I_{k,00,r}), & 0 \le k < S, \\ (2)\ P(C_{k,00,r-1}), & k = S, \\ (3)\ P(C_{k,00,r-1}) - P(O_{k,00,r}), & k > S. \end{cases}$$

$$P(A_{k,10,r}) = \begin{cases} (4)\ P(C_{k,10,r-1}) + P(I_{k,00,r}), & 0 \le k < S, \\ (5)\ P(C_{k,10,r-1}) - P(O_{k,10,r}), & k \ge S. \end{cases}$$

$$P(A_{k,01,r}) = \begin{cases} (6)\ 0, & 0 \le k < S, \\ (7)\ P(C_{k,01,r-1}) + P(O_{k+1,00,r}), & k \ge S. \end{cases}$$

$$P(A_{k,11,r}) = \begin{cases} (8)\ 0, & 0 \le k < S-1, \\ (9)\ P(C_{k,11,r-1}) + P(O_{k+1,10,r}), & k \ge S-1. \end{cases}$$

Proof. By Lemma 1 (1), an overcrowded bucket $b[i]$ of type 00 (resp. 10) before step 1 of round r becomes of type 01 (resp. 11) after step 1 of round r. This explains equations (3) and (7) (resp. (5) and (9)). By Lemma 1 (2), a non-full bucket $b[i+1]$ of type 00 before step 1 of round r becomes of type 10 after step 1 of round r. This explains equations (1) and (4). The other equations are self-explanatory. □

The probability for spill-in and spill-out at round r, or $P(O_{k,00,r})$, $P(O_{k,10,r})$, and $P(I_{k,00,r})$, can be computed in the following way.

Lemma 3. *In step 1 of round r,*

$$P(O_{k,i0,r}) = \frac{P(C_{k,i0,r-1}) \sum_{j=0}^{S-1} P(C_{j,00,r-1}) \sum_{j=0}^{\infty} P(C_{j,i0,r-1})}{(\sum_{j=0}^{\infty} P(C_{j,00,r-1}) + \sum_{j=0}^{\infty} P(C_{j,01,r-1})) \sum_{j=0}^{S} P(C_{j,i0,r-1})}$$

$$P(I_{k,00,r}) = P(C_{k,00,r-1}) \frac{(\sum_{j=S+1}^{\infty} P(C_{j,00,r-1}) + \sum_{j=S}^{\infty} P(C_{j,10,r-1}))}{(\sum_{j=0}^{\infty} P(C_{j,00,r-1}) + \sum_{j=0}^{\infty} P(C_{j,10,r-1}))}$$

Let X_r be the random variable representing the number of excessive keys in a bucket after step 1 of round r. Then its expected number is calculated as follows. The expected value of X_r, the number of excessive keys in a bucket, after step 1 of round r is as follows.

Lemma 4. *After step 1 of round r,*

$$E(X_r) = \sum_{k=S+1-i}^{\alpha S b} \sum_{0 \le i,j \le 1} (k - S + i) P(A_{k,ij,r})$$

Now we analyze the step 2 of round r. By collecting all the excessive keys from overcrowded buckets, $P(B_{k,ij,r})$ for $k \ge -1$, $i, j \in \{0, 1\}$, and $r \ge 1$ is as follows where $P(B_{-1,ij,r})$ is the probability that a bucket does not participate in the step 3 of round r.

Lemma 5. *At the end of step 2 of round r,*

$$P(B_{k,0j,r}) = \begin{cases} P(B_{k,0j,r-1}) + \sum_{i=S+1}^{\infty} P(A_{i,0j,r}), & k = -1, \\ P(A_{k,0j,r}), & 0 \le k \le S, \\ 0, & k > S. \end{cases}$$

$$P(B_{k,1j,r}) = \begin{cases} P(B_{k,1j,r-1}) + \sum_{i=S}^{\infty} P(A_{i,1j,r}), & k = -1, \\ P(A_{k,1j,r}), & 0 \le k \le S-1, \\ 0, & k > S-1. \end{cases}$$

The Poisson parameter λ_r, or the average number of keys distributed to a non-full or full bucket in round r, is as follows.

Lemma 6. *For step 3 of round r,*

$$\lambda_r = \frac{E(X_r) S}{\sum_{k=0}^{S-1} P(B_{k,00,r}) + P(B_{S,00,r}) + \sum_{k=0}^{S-2} P(B_{k,10,r}) + P(B_{S-1,10,r})}$$

Now we compute the probabilities of the events at step 3 of round r. $P(C_{k,00,0})$, the probability that a bucket has k non-spilled keys after round 0, is the same as $P(A_{k,0})$ and $P(\lambda_0, k)$, thus $P(C_{k,00,0}) = P(A_{k,0}) = P(\lambda_0, k)$. $P(C_{k,ij,r})$ for $r \ge 1$ are calculated as follows.

Lemma 7. *After step 3 of round r*

$$P(C_{k,00,r}) = \begin{cases} \sum_{j=0}^{k} P(B_{j,00,r}) P(\lambda_r, k-j), & 0 \le k < S, \\ \sum_{j=0}^{S} P(B_{j,00,r}) P(\lambda_r, k-j), & k \ge S. \end{cases}$$

$$P(C_{k,10,r}) = \begin{cases} \sum_{j=0}^{k} P(B_{j,10,r}) P(\lambda_r, k-j), & 0 \le k < S-1, \\ \sum_{j=0}^{S-1} P(B_{j,10,r}) P(\lambda_r, k-j), & k \ge S-1. \end{cases}$$

$$P(C_{k,01,r}) = \begin{cases} 0, & 0 \le k < S, \\ P(B_{k,01,r}), & k \ge S. \end{cases}$$

$$P(C_{k,11,r}) = \begin{cases} 0, & 0 \le k < S-1, \\ P(B_{k,11,r}), & k \ge S-1. \end{cases}$$