Revisiting Hash Table Design for Phase Change Memory

Biplob Debnath, Alireza Haghdoost *, Asim Kadav,
Mohammed G. Khatib † and Cristian Ungureanu ‡
NEC Laboratories America

Abstract

Phase Change Memory (PCM) is emerging as an attractive alternative to Dynamic Random Access Memory (DRAM) in building data-intensive computing systems. PCM offers read/write performance asymmetry that makes it necessary to revisit the design of in-memory applications.

In this paper, we focus on in-memory hash tables, a family of data structures with wide applicability. We evaluate several popular hash-table designs to understand their performance under PCM. We find that for write-heavy workloads the designs that achieve best performance for PCM differ from the ones that are best for DRAM, and that designs achieving a high *load factor* also cause a high number of memory writes. Finally, we propose PFHT, a PCM-Friendly Hash Table which presents a cuckoo hashing variant that is tailored to PCM characteristics, and offers a better trade-off between performance, the amount of writes generated, and the expected load factor than any of the existing DRAM-based implementations.

1. Introduction

Phase Change Memory (PCM) is an emerging class of non-volatile memory technologies that offers an attractive combination of features, making it a likely replacement for DRAM. PCM can be built at higher chip densities than DRAM, which is expected to translate into higher capacities and lower cost per GB. It does not require power to retain information, which not only means that it can be used for persistent storage but also leads to lower power and energy consumption when idle. Its read access latency is close to that of DRAM, while its write latency is about 3-12 times higher [19, 35]. It also suffers from a limited write endurance. For example, a PCM cell can only be written about 10^8 times, which is a relatively low number given its low write access latency.

Recent research has investigated: techniques to develop operating system components such as file systems to use PCM instead of the more traditional block-based storage (flash or hard-disk) [8]; techniques to provide the applications a high-level API with durability and consistency guarantees [7, 35]; techniques to provide crash consistency through versioning [33]; hardware techniques to mitigate the endurance problem of PCM through wear-leveling [30]. While one can apply these generic techniques to reduce wear and improve performance, different underlying algorithms will exhibit different locality of memory accesses or a different mixture of read-write operations. We believe that even with wear-leveling, algorithms should avoid excessive writes in order to improve endurance of PCM. Hence, a question still remains: *what algorithms should one use that would work best given the unique properties of PCM?* In this paper, we attempt to answer this question for *hash tables*.

Hash tables are data structures that can map keys to values, and are widely used in computer systems. They form the basis of many NoSQL storage systems [1, 13, 23], are used by databases to provide high-speed access to their working sets [5, 10, 18], or used to implement in-memory caching systems like Memcached [15, 27]. In such systems, the hash tables often occupy hundreds of gigabytes and reach limits imposed by DRAM capacity, cost, or even power costs [4]. As such, they are likely to benefit from using PCM instead of DRAM. We note that in many cases these applications require a high rate of in-memory updates: real-time analytics such as serving ads [26], caching when the miss rates are high, and hash-based joins. For such uses, a high amount of writes to memory can adversely affect hash table performance and PCM endurance.

In this paper, we evaluate hash table performance over DRAM and PCM. For our evaluation, we choose a basic hash table design (linear probing) and many recent memory-efficient designs such as cuckoo hashing [28] and hopscotch hashing [17] that attempt to reduce the number of cache misses with guaranteed lookup times. Since we want to investigate the differences in hash table behavior between DRAM and PCM, we focus on designs that generate a different mixture of read-write memory accesses and do not

* Work done as NEC Labs intern. Now at University of Minnesota.

† Work done at NEC Labs. Now at HGST.

‡ Work done at NEC Labs. Now at Google.

INFLOW'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3945-2/15/10. . . $15.00.
http://dx.doi.org/10.1145/2819001.2819002

consider techniques to improve CPU utilization by reducing branch operations [36] and utilizing SIMD instructions [31].

We find that the memory efficient hash table designs suffer from the problem of *cascading writes*. Cascading writes happens when inserting a single key-value (KV) pair triggers a chain of displacements of existing KV pairs, and results in a large number of writes. Since PCM suffers from read-write asymmetry in performance, cascading writes result in significantly degraded performance during insert operations.

We find several interesting results and the contributions of this paper are as follows. First, we find that because of the read-write asymmetry, for many workloads the design that achieves the best performance for PCM differs from the one that is best for DRAM. Second, for hash table designs such as cuckoo hashing and hopscotch hashing, that can support high load factor, we quantify how much these hash tables suffer from the problem of *cascading writes* that can hurt their performance and endurance of PCM. Based on the insights gained through these experiments, we propose a hash table design, PFHT(stands for PCM-Friendly Hash Table), that takes into account the specific properties of PCM and offers a better trade-off with respect to performance, amount of writes, and load factor compared with existing DRAM-based designs.

2. Background

Hash tables provide associative array functionality by storing *key-value* pairs at specific locations which are determined by applying one or more *hash functions* to the key.

2.1 Hash Table Designs

A basic hash table design is *standard chained hashing*, or *linear chaining*. It uses an array of *buckets* that holds zero or more key-value (KV) pairs in a linked-list, and a single hash function that is applied to a key to determine the index of the array containing the bucket that may hold the key and its associated value. Although easy to implement, chained hashing has two important drawbacks: poor CPU cache locality due to pointer traversal (necessary when buckets have more than one KV pair) and heap allocation and de-allocation overhead for workloads with a high rate of modifying operations (inserts and removes). In addition, linear chaining is often criticized for its simplistic handling of collisions which is prone to adversarial attacks to adversely affect hash table performance [9].

Array hashing [2] is a variant of standard chained hashing that allocates contiguous heap space to store a bucket, using *realloc* to handle the fluctuation in the number of elements in a bucket. Although this design avoids pointer chasing, it is unattractive for workloads with high number of insert and delete operations because realloc is expensive.

Linear probing uses an array of buckets to store KV pairs and avoids heap operations. During a KV pair insert, the key is hashed to an array index and stored there if the bucket is empty. In case it is occupied, subsequent indices are scanned until an empty bucket is found. Linear probing is CPU cache friendly as all KV pairs are accessed sequentially. However, as the load factor increases, the number of buckets traversed to find an empty bucket also increases. For example, at a load factor of 0.90, about 50 buckets need to be traversed to find an empty bucket [22]. A further complication comes from the way deletions are handled: simply marking a bucket as free will incorrectly stop the search for a KV pair that is inserted further out, leading to a poorer lookup performance for workloads with frequent deletions.

Cuckoo hashing [28], like linear probing, uses an array of KV pair buckets but puts a bound on the number of buckets inspected to find a KV pair. Each KV pair can be placed in any one of two buckets. Cuckoo hashing uses two separate hash functions to compute these two separate bucket addresses. As a result, these buckets are not adjacent. Each bucket can accommodate one KV pair. During insert, if both buckets are occupied, a KV pair in one of the two buckets is randomly chosen and displaced to its other bucket to make room for the incoming KV pair. Furthermore, the displaced KV pair may displace another KV pair if both its buckets are full creating a chain of displacements. This process continues until no further displacement is required (i.e., an empty bucket is found) or the upper bound for the number of displacements is reached. When no empty bucket is found for a displaced KV pair, the hash table is declared as *full* and resizing is needed. Kirsh et al. [21] propose using additional storage, called as *stash*, to store the overflow KV pairs, to avoid resizing. Some variants of cuckoo hashing use more than two alternative buckets for each KV pair [16], and other variants use larger buckets in order to contain multiple KV pairs [11, 12, 29].

Hopscotch hashing [17] is an improvement over cuckoo hashing for its improved cache utilization and concurrency. In hopscotch hashing, the all possible locations for a specific key are contiguous. The number of these locations is called as the *hop distance*. Hop distance is the maximum number of locations inspected during lookup. The hop distance is usually chosen to be a multiple of the cache line size, making hopscotch CPU cache friendly. To accelerate lookups, each bucket also maintains a bitmap which specifies all the relevant positions in the neighborhood that need to be examined to locate an item. During inserts, the hopscotch algorithm hashes to a specific location based on the key. It then uses linear probing to locate an empty slot. If the empty slot is out of the hop range, an attempt is made to bring that empty position within the hop range by displacing other items without violating their hop constraints. If such a displacement is not possible, the hash table becomes full and needs to be resized.

Summary. To summarize, chained hashing uses linked lists to store colliding items. However, its use of dynamic mem-

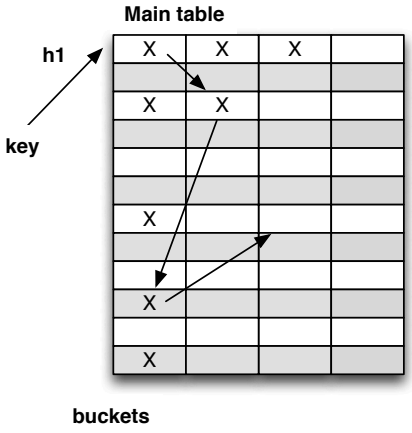

Figure 1. Cascading write effect in cuckoo hashing. During insert, a KV pair can kick out an existing KV pair, and moves it to an alternate location, triggering a chain of inserts.

ory allocation and pointer accesses causes lots of CPU cache misses, degrading performance. Array hash uses contiguous memory to avoid the pointer chasing problem. However, it still suffers from the overhead associated with frequent *realloc* calls. Linear probing improves CPU cache utilization by avoiding dynamic memory allocation and accessing items in consecutive cache lines. However, as the load factor increases, inserting a single KV pair requires traversing large number of cache lines to find an empty slot. Memory efficient hash tables such as cuckoo hashing and hopscotch hashing do not require dynamic memory allocation and support high load factor with bounded lookup times. However, these hash tables suffer from the problem of *cascading writes*.

2.2 The Problem of Cascading Writes

By *cascading writes*, we mean that to insert a KV pair, we may need to displace existing KV pairs, thus incurring multiple write operations for a single insert. Figure 1 shows cascading write problem for cuckoo hashing. During an insert, a single KV pair, displaces existing KV pairs which triggers further displacements until an empty bucket is located. Hopscotch hashing also suffers from cascading writes because it may trigger a series of displacements to insert a new KV pair.

In DRAM-based systems, there is a symmetry between read-write performance. As a result, the performance of hash tables over DRAM is dominated by CPU cache utilization. However, for PCM-based systems, the read-write performance asymmetry imply that the number of writes additionally affects hash table performance. Cascading writes, which is caused by KV pair displacements during inserts for memory-efficiency can slow down inserts and wears off PCM quickly.

Table 1 shows a comparison of existing hash table designs. In terms of cascading writes, chaining, array hash and linear probing perform better (especially when the load factor is high), while cuckoo [28] and hopscotch [17] hashing perform worse. In terms of CPU cache utilization, chained and cuckoo hashing perform worse while linear probing and hopscotch perform better. Our goal is to design a hash table which achieves high CPU cache utilization and avoids the cascading write during inserts.

Collision Resolution Strategy	CPU Cache Utilization	Cascading Writes	Frequent Memory Allocation
Chaining	Low	No	Yes
Arrayhash	High	No	Yes
Linear Probing	Medium	No	No
Cuckoo	Low	Yes	No
Hopscotch	High	Yes	No
PFHT (our design)	High	No	No

Table 1. Comparison of different hash table designs (at high load factor)

3. Hash tables over DRAM vs. PCM

In this section, we evaluate the memory efficiency and performance of commonly used hash tables. Linear chaining and array hash perform poorly under memory pressure due to frequent `alloc()` and `realloc()` calls, respectively. Hence, we only characterize linear probing, cuckoo and hopscotch hashing designs over DRAM and PCM. In Section 4, we characterize PFHT performance.

3.1 Evaluation Setup

We run all our experiments on a 6-core Intel Xeon 2.0 GHz machine with 6MB CPU cache and 24GB of DRAM memory. We evaluate all hash table designs on PCM using the Mnemosyne framework [35]. Mnemosyne emulates PCM hardware by introducing a delay to emulate slower writes to DRAM by wrapping hardware access macros. It adds a fixed amount of delay that hits DRAM from the cache. For our experiments, we emulate an extra delay of 150 ns for write operations [35]. We modify our hash table implementations to allocate PCM memory from mnemosyne's library. We use mnemosyne's vista heap allocator and store hash table data in persistent memory (using mnemosyne's *pmalloc()* calls), and wrap hash table operations using mnemosyne's transactions. This makes the hash table persistent. We also make modifications to mnemosyne to obtain the total number of PCM writes (including log writes) performed.

3.1.1 Workloads

For evaluation, we have implemented all the hash tables for a fixed size key-value (KV) pair workload with key size of 8 bytes (i.e., 64 bits) and value size of 8 bytes. Our implementation can be easily extended to support variable-length KV pairs. We can generate a 64-bit long checksum from a key, and use this checksum as hash key. Next, we can use the 64-bit address for each KV pair as a hash value. During lookups, if the checksum matches

then we need to compare with the full key to determine a successful hit. Here, we use the following two workloads.

Write-Once (WO) workload. For this workload, we create an empty hash table and perform bulk insert of KV pairs to reach target load factor of 90% and 95%. WO workload represents use cases, where hash tables are used mostly for lookup operations. For example, facebook [3], spotify [32] and the intermediate stage of hash-join algorithms [6] of various database systems run such workloads.

Write-Heavy (WH) workload. For this workload, we first use WO workload to reach a target load factor. Then, we delete an existing KV pair and insert a new KV pair continuously. Hence, even though the overall load factor remains fixed, the hash table is subjected to continuous random inserts and deletes. WH workload represents the use cases incurring frequent insertions and deletions. For example, metadata caches and analytics workloads (i.e., advertising servers [26]) exhibit characteristics similar to the WH workload.

For both these workloads, we perform lookup operations for existing and non-existing keys (also called as successful and unsuccessful lookups). In the rest of the paper, we show performance results for a hash table sized for 25 million KV pairs. We use 16 bytes for every KV pair, so for the entire hash table, we need at least 381.5 MB of memory (apart from any metadata overhead). The machine we use for evaluation has 6 MB of CPU cache. Since, we are evaluating the CPU cache utilization and memory write-behavior (for PCMs), using a larger hash table do not affect our results. In addition to above two workloads, we have used a realistic caching workloads in Section 4.6.

3.1.2 Hash table Configurations

Linear probing inserts a new KV pair in the next available slot. This may result in very long lookup times. However, it can achieve 100% load factor. Cuckoo and hopscotch hash table designs can only achieve limited load factor since they guarantee an upper bound on the lookup times. The load factor achieved by these hash tables is dependent on different hash table parameters that we describe below.

With hopscotch hashing, increasing the hop range will improve the load factor of the hash table since one can hash a large number of items for a specific key. Hence, we pick a large hop range of 64 because any smaller (such as 32) cannot achieve greater than 90% load factor. We use a 64-bit (unsigned long) variable to store this hop range. We find that even with a hop range of 64, we cannot achieve very high load factors (greater than 95%). However, we limit ourselves to a hop range of 64 to avoid any additional write overheads for updating more than one variable. To address this issue and accommodate higher load factors, like past work [21], we augment hopscotch hashing with a small *stash*, which is a very small hash table to store any overflow entries. For 90% load factor, we use a fixed size stash of 8192 KV

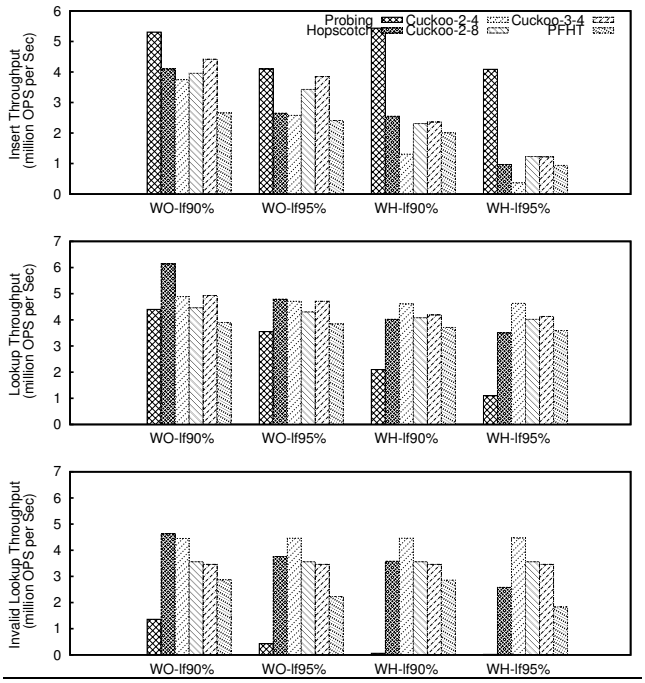

Figure 2. The insert, successful lookup, unsuccessful lookup performance for hash tables running on DRAM for write-once (WO) and write-heavy (WH) workloads.

pairs. However, for the 95% load factor we use 3% of the hash table size as stash. These stash sizes are empirically determined by running various configurations (not shown in the paper).

With cuckoo hashing, increasing the number of alternative buckets and bucket size helps to achieve a higher load factor. However, this increases the CPU cache misses and hurts the lookup performance. To restrict our search space of possible cuckoo hash configurations, we evaluate cuckoo hashing with two buckets and a bucket size of single cache line as the base design. This is a commonly used cuckoo configuration over DRAM for good performance [14]. One KV pair size is 16 bytes, and each cache line is 64 bytes long. So, we can fit 4 KV pairs in each cache line. To evaluate the impact of bucket sizes, we use two cache lines (i.e., each bucket can contain 8 KV pairs); and to evaluate the impact of increasing alternative buckets, we use three buckets. In all cases, we set maximum displacement chain length to 512, and we do not need to use any stash (which is consistent with the results reported by Erlingsson et al. [12]). In the rest of this paper, we denote above three cuckoo configurations as *cuckoo-2-4*, *cuckoo-2-8*, and *cuckoo-3-4*, respectively.

3.2 Performance Over DRAM

Figure 2 shows that linear probing exhibits the best insert performance as it does not suffer from cascading write problem. However, it performs poorly for the invalid lookups, because there is no upper bound in the number of KV pairs to be scanned. Cuckoo and hopscotch hashing exhibit better

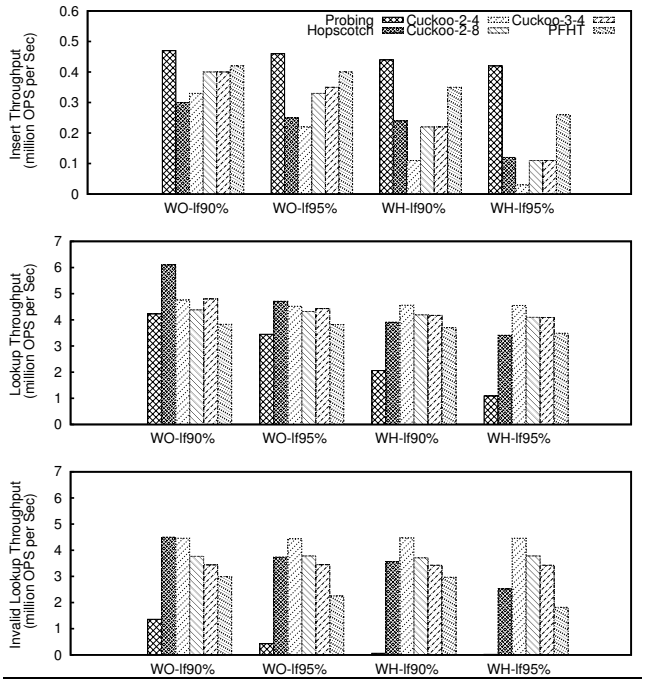

Figure 3. Insert, successful lookup, and unsuccessful lookup performance for hash tables running on PCM for write-once (WO) and write-heavy (WH) workloads.

lookup performance by using multiple alternative locations for each KV pair. However, for higher load factor, these hash tables displace KV pairs to make space and the problem of cascading writes hurts performance. This problem becomes severe for the write-heavy workloads, and its performance degrades.

Overall, we find that for write-heavy workloads with higher load factor, cuckoo hashing performs better than hopscotch hashing over DRAM. Next, we evaluate how expensive are the excessive cascading writes when these hash tables are used over PCM, since PCM suffers from read-write performance asymmetry.

3.3 Performance Over PCM

Figure 3 shows that the performance for successful and unsuccessful lookups is similar to DRAM. This is expected because PCM exhibits similar read latency as that of DRAM. However, we find that using PCM as memory can hurt insert performance for three reasons. First, there is an extra latency introduced by writing to PCM. Second, executing hash table operations as transactions for durability introduces additional delay to write the redo log and a commit record. Finally, we find that allocating persistent memory is very slow and frequent memory allocations can add additional latency because allocating memory introduces additional transactions. This especially hurts linear chaining and array hashing schemes because they require frequent memory allocations.

With inserts, we find that linear probing has the best performance over PCM. The reason is that linear probing requires the least number of writes per insert. Inserting an entry only requires storing the KV pair in the array bucket. Both cuckoo and hopscotch hash tables have considerably slower insert performance as compared to DRAM (and linear probing). Apart from the delay introduced by the read-write asymmetry, we find that cascading writes introduce extra latency during inserts for both hash table designs. In the write-once case, we find that cuckoo configurations perform better than hopscotch hashing configurations. For the write-heavy case (in the 90 and 95% load factor), we find that the cascading write effect becomes severe with cuckoo. As a result, hopscotch hashing outperforms cuckoo configurations.

Hashing	WO-lf-90%	WO-lf-95%	WH-lf-90%	WH-lf-95%
Probing	4.0	4.0	4.0	4.0
Hopscotch	6.6	7.4	7.2	11.4
Cuckoo-2-4	4.7	5.8	9.3	26.7
Cuckoo-2-8	4.3	4.6	5.7	8.8
Cuckoo-3-4	4.2	4.5	5.7	8.6
PFHT	4.0	4.1	4.7	5.9

Table 2. PCM write count per insert operation

Write Count. Table 2 shows the write count per insert operation. Linear probing incurs 4 writes per insert. These writes include inserting the KV pair, updating any associated metadata (like bitmaps for hopscotch), and any additional writes for durability (transaction commit). We see that for achieving high load factor, cuckoo and hopscotch hash tables displace multiple KV pairs. As a result, they suffer from poor performance because of cascading writes. For example, at 95% load factor, we find that cuckoo-2-4 (the configuration used by Fan et al. in memc3 [14]), incurs on an average 26.7 writes per insert, while hopscotch hashing, which is more write efficient, incurs about 11.4 writes per insert.

Summary. Overall, for write-heavy workloads with high load factor, we find that hopscotch hashing performs better than cuckoo hashing over PCM, which differs from the DRAM results. The reason is that cuckoo hashing suffers severely from cascading writes that makes it slower over PCM.

3.4 Memory Consumption

Table 3 shows the overall memory consumption for different hash tables. Linear probing and cuckoo need 16 bytes per KV pair, while hopscotch hash table (without the stash) requires 24 bytes. The 64-bit bitmap used in hopscotch implementation to improve performance incurs 50% memory overhead. Hopscotch hash table consumes additional memory for stash. Overall, we find that hopscotch uses 24 bytes per KV pair for the 90% load factor workload and 26 bytes for the 95% load factor workload. Hence, hopscotch consumes 50% more memory than cuckoo and linear probing hash tables.

In the next section, we describe how we can design a memory efficient hash table that does not suffer from unbounded lookup performance like linear probing, while offering benefits of multi-hash designs like the cuckoo.

Hashing	lf=90%	lf=95%
Probing	16.00	16.00
Hopscotch	24.01	26.15
Cuckoo-2-4	16.00	16.00
Cuckoo-2-8	16.00	16.00
Cuckoo-3-4	16.00	16.00
PFHT	16.00	18.09

Table 3. Memory used per KV. Hopscotch and PFHT use 3% chaining-based stash for the 95% load factor; therefore, memory usage becomes higher.

4. PCM-Friendly Hash Table Design

To address the limitations of existing hash tables over PCM, we propose a hash table configuration for PCM called the PCM-Friendly Hash Table (PFHT). PFHT is a hash table designed to limit PCM writes and improve write performance.

4.1 PFHT Design

PFHT is a cuckoo hashing variant that offers the advantages associated with cuckoo hashing i.e., a memory efficient design for high load factor and guaranteed lookup times. However, PFHT avoids the problems of cascading writes which affects cuckoo and hopscotch hashing. It has two components – the *Main table* and the *stash*, as shown in Figure 4. The main table (MT) provides good CPU cache performance and avoids cascading writes while the stash helps PFHT to achieve a high load factor.

Main Table: Main Table (MT) is the primary storage for the PFHT. MT is a two-dimensional array of buckets. The buckets are stored contiguously in memory. Each bucket contains a fixed number of slots to store KV pairs. PFHT uses two buckets to store any incoming KV pair. Furthermore, each bucket fits in one or multiple consecutive cache lines. Hence, the number of slots in a bucket depends on the cache line size, which is fairly large (64 bytes) in modern processors.

Stash: Stash is a small auxiliary storage that PFHT uses to store any KV pairs that it fails to insert in MT. Most of the cuckoo hashing designs allow higher (or unbounded) number of KV displacements in MT during inserts, and suffer from the problem of cascading writes during inserts. To avoid this problem, PFHT limits KV displacement in MT to just one. However, this reduces the load factor achieved by PFHT. Hence, PFHT may fail to insert KV pairs in MT more frequently than cuckoo. To offset the load factor loss in MT, due to the limited shuffling of KV pairs, PFHT uses the additional stash space. PFHT has the following properties that make it an attractive choice for PCM.

- **Load balanced inserts**: A single KV pair can be stored in two possible buckets in MT. PFHT picks the least loaded bucket to store a KV pair and keeps the MT load-balanced. In contrast, cuckoo hashing selects a bucket randomly to insert a KV pair. This scheme does not have adverse affects for DRAM even with an unbounded movement chain length [14]. However, the extra writes may hurt PCM performance. By keeping the buckets load-balanced, PFHT ensures that extra writes are not required to store new KV pairs.
- **No cascading writes**: PFHT avoids cascading write effect by allowing atmost one displacement during inserts. During an insert, if both the MT buckets are full, cuckoo hash continuously displaces any one of the existing KV pairs and causes cascading writes. Instead, PFHT, checks if it is possible to move any KV pair from one of its two buckets to its alternative location. If no movement is possible, PFHT inserts the new KV pair into the stash.
- **High CPU cache utilization**: PFHT stores keys that hash to the same location in contiguous buckets, improving CPU cache performance. PFHT uses larger buckets to achieve high load factor. However, using large size bucket increases lookup latency. Therefore, to achieve a good balance between load factor and lookup latency, PFHT sets the bucket size to two cache lines.

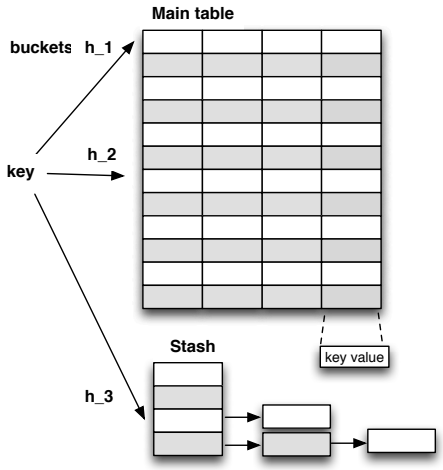

Figure 4. The figure shows PFHT design. PFHT stores key-values in two bucketed arrays in the main table (MT). An additional stash stores overflow KV pairs for improved load-factor. Each row in the MT occupies multiple consecutive cache lines.

4.2 PFHT Operations

We now describe how PFHT performs KV pair insertion, lookup and deletion operations.

1. *Insert*: First, two hash functions map a KV pair to two possible buckets in MT. The least loaded bucket is chosen (with ties broken randomly) to store a KV pair. If both buckets are full, PFHT tries to re-insert an existing KV pair from any of the two buckets to its alternative location. If successful, PFHT inserts the KV pair in MT. Otherwise, it inserts the new KV pair into stash.

2. *Lookup*: To lookup a key, PFHT scans the two MT bucket locations that the key hashes to. If the key is not found, PFHT looks up the stash to locate the key. If the key is found, PFHT returns the associated value, otherwise returns $null$.
3. *Delete (or update)*: To delete (or update) a KV pair, the KV pair is located using the lookup operation. If key is found in the MT, the KV pair is deleted (or updated) in place. If the corresponding KV pair is found in the stash, the delete (or update) follows the stash implementation.

Stash Size. We determine the stash size empirically. We set stash to x% and MT to 100-x% of the total hash table size and record the load factor that can be supported without dropping any entries. We find that for load factors up to 90%, a fixed stash of 8192 entries is sufficient while for 95% load factor, we need 1-5% stash. In addition, the stash size (i.e., x%) does not vary with the change of hash table size. We use 3% stash for our evaluation.

We use chaining based hash table as stash. Chaining and array hash may not be suitable for PCM as they cause frequent memory allocation and de-allocation to insert and delete KV pairs and have very high memory overhead. In general, array hash is worse because it frequently calls *realloc* calls. Since, PFHT uses a very small stash (3%) compared to the MT, using a linear chaining based stash implementation has negligible performance overheads. It is discouraged to use linear probing as a stash implementation because it has poor lookup performance. Furthermore, we do not recommend to use cuckoo and hopscotch hash configurations for stash because they may cause cascading writes.

4.3 Write-once Workload Performance

We now compare PFHT with other hash tables and report our results in Figure 3. We find that PFHT limits the number of writes to PCM, and provides good performance.

For the insert performance, we find that linear probing performs best, followed by PFHT, cuckoo, and hopscotch hashing. Linear probing inserts a KV pair in the next available bucket which improves insert times but results in poor lookup performance. For multi-hash designs, PFHT performs the best, followed by cuckoo and hopscotch in the write-once workload. For write-once workload, cascading write effect is low, since KV pairs are inserted in one-shot, up to the required load factor without any deletions. Hopscotch always pays the fixed additional cost of updating the bitmap that results in high average write counts that hurts performance. Overall, we find that PFHT has a good write performance owing to the low number of PCM writes as seen in Table 2. For higher load factors, this gap increases since PFHT performs limited writes. Hopscotch and cuckoo 2-4 configurations only perform 75% that of PFHT write performance. In terms of number of writes, linear probing performs best, followed by PFHT.

For successful and unsuccessful lookups in the write-once case, both hopscotch and cuckoo outperform PFHT. Although cuckoo-2-8, and PHFT use two hash buckets with a bucket size of two cache lines, PHFT performance is lower because it uses a stash to store overflow KV pairs. The stash needs to be accessed during lookups to determine whether a KV pair exists. During lookup if a KV pair is found in the main table, PHFT does not access the stash. However, the stash will always be accessed for the non-existing KV pair lookups. Therefore, unsuccessful lookup throughput is lower than the successful lookup throughput.

4.4 Write-heavy Workload Performance

We find that PFHT is significantly faster (more than 2X as compared to hopscotch and more than 8X as compared to cuckoo 2-4 configuration) for inserts. Linear probing's insert performance is adversely affected by deletes and inserts since the maximum probe length is effectively set to the length of the entire array. Hence, linear probing performs 50% slower for valid lookups and 20X slower for invalid lookups.

For invalid lookups, we find that linear probing is very slow. We also see that cuckoo hash tables and hopscotch perform about 10% better with successful lookups as compared to PFHT. In addition, PFHT is 20-50% slower with invalid lookups than cuckoo and hopscotch hashing. Hence, we see that PFHT offers an new design point in the trade-off space, by offering a higher write throughput with lower lookup (especially invalid lookup) performance that can be useful for write-heavy workloads like ad-servers.

4.5 Writes and Memory Consumption

Overall, as seen in Table 2, we find that PFHT incurs on average 5.9 writes per insert for the write-heavy workload at 95% load factor. Cuckoo hash causes more than $>$ 450% writes with 26.7 writes per insert, and hopscotch causes about 190% as many writes with 11.4 writes per insert. Even at lower load factor of 90%, we see that PFHT with 4.7 writes per insert causes the most few writes amongst the multi-hash hash table implementations. As mentioned earlier, linear probing represents the base case with no shuffling involved (4 writes/insert). We find that PFHT only incurs about 6 writes in write-heavy case and close to 4 writes/insert (baseline case) in the remaining configurations.

Hence, we find that PFHT design avoids cascading writes. The memory consumption of hash tables does not change with the underlying memory device. As observed with DRAM performance, hopscotch hash configurations consume 50% more than cuckoo hashing. However, we do see that PFHT consumes memory comparable to cuckoo hashing with 90% load factor and additional 2 bytes per KV pair with 95% load factor.

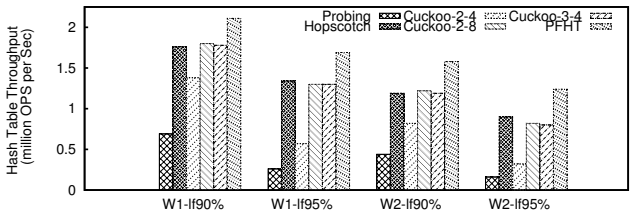

Figure 5. Throughput in million operations per second for different hash table implementations over PCM. Here, W1 represents 95% hit ratio workload while W2 represents 90% hit ratio workload.

4.6 Real Workload Performance

One of the most straightforward applications of PCM is to use it as a key-value cache since PCM can store large amounts of data at lower power cost as compared to DRAM [19]. To evaluate how PFHT and other hash tables perform for caching workload scenarios, we run simulated workloads with a cache hit rate of 90% and 95%. In the first case, 10% of the lookups are invalid and hence incur cache misses. In the second case, there are 5% cache misses. During a cache miss, we randomly delete (evict) a KV pair from the hash table and re-insert the missing KV pair.

Figure 5 shows hash table throughput in million operations per second for a cache over PCM. Linear probing has poor lookup performance especially for invalid lookups, and the overall throughput is very low. We find that PFHT outperforms both hopscotch and cuckoo hashing. As the load factor increases, the throughput decreases because of high number of write operations. Hence, with higher miss rates, there are more delete and insert operations to add new KV pairs in the cache, and consequently results in high number of write operations. Hence, we find that PFHT performs at 5X throughput over cuckoo hashing and 1.4X throughput over hopscotch hashing, and gives good performance for real workloads.

5. Related Work

Application support for PCM: PCM-DB [6] and related systems [34] explore database algorithms such as join and sort designs that limit the number of writes. These systems can benefit from PCM optimized hash tables such as PFHT often used as a store for database processing such as data structures for query processing, logging and recovery.

Hash table improvements: Recent work in hash tables encourages memory efficient hash tables [14, 25] but these designs may result in large number of writes. In addition, Li et al. [24] suggest several variants of cuckoo hashing to improve the concurrency. Zukowski et al. [36] and splash table [31] reduce branch operations in modern processors in order to improve lookup performance. Our PFHT design is complementary to these designs.

Other cuckoo variants Past work [21], has looked at using a small constant-size stash (i.e., 3-4 KV pairs) with cuckoo hash to avoid rehashing. However, this stash is rarely used because there is no limit on the displacement chain length during inserts. PFHT is complementary to the cuckoo hashing for hardware implementation proposed by Kirsh et. al [20]. This design works well for workloads having no delete operations. In contrast, PFHT does not impose any restriction in the movement direction and is more suitable for write-heavy workloads, since it can support higher number of entries in the main table.

6. Discussion and Conclusion

With PCM, the number of writes introduce additional design constraint apart from performance and load factor. We find that existing hash tables, such as cuckoo hashing, are designed for DRAM and are oblivious to PCM write characteristics. In addition, hash tables that guarantee bounded lookup times and high load factor suffer from the problem of cascading writes, which affects PCM performance and endurance. Based on our experience, we characterize the following best practices for write-heavy workloads:

- *Large bucket sizes*: Using large bucket sizes improves the number of KV pairs that can be stored in a single hash table row. However, bucket sizes need to be cache-line aligned. If the buckets sizes are too large, lookup and insert operations may result in fetching of multiple cache lines and hurt performance.
- *Multi-hash functions*: Using multiple hash functions improves the write behavior since there are multiple slots where a single KV pair can be stored. However, using large number of functions increases the average number of places to locate a KV pair and hence increases the lookup latency.
- *Use of a small stash*: A stash stores a very small percentage of hash table entries (1-5%) and is used only when the load factor is high. Hence, instead of increasing the number of hash functions one can add a small stash, that will only be used for load factors $> 90\%$. This allows bounded lookup times since an empty stash is not used during lookup.
- *Balanced design*: Having a load balanced design ensures that hash table occupancy is uniform. Instead of shuffling elements during high occupancy and causing unpredictable performance, PFHT pays an incremental cost in every insert to carefully insert KV pairs in the least loaded buckets.

Based on these lessons, we propose PFHT that considers the read-write asymmetry of PCM. PFHT avoids the problem of cascading writes in hash tables and provides good performance under the write heavy workloads. Finally, we demonstrate our results on PCM emulator, these results may apply to other storage class memories that have read-write performance asymmetry.

References

[1] 10 GEN INC. Mongo-DB: Open source document database. http://www.mongodb.org/.

[2] ASKITIS, N. Fast and Compact Hash Tables for Integer Keys. In *ACSC* (2009).

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *SIGMETRICS* (2012).

[4] BARROSO, L. A., AND HÖLZLE, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture 4*, 1 (2009).

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[6] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking Database Algorithms for Phase Change Memory. In *CIDR* (2011).

[7] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *ASPLOS* (2011).

[8] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-Addressable, Persistent Memory. In *SOSP* (2009).

[9] CROSBY, S. A., AND WALLACH, D. S. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium* (2003).

[10] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD* (2013).

[11] DIETZFELBINGER, M., AND WEIDLING, C. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science 380*, 1-2 (July 2007).

[12] ERLINGSSON, U., MANASSE, M., AND MCSHERRY, F. A Cool and Practical Alternative to Traditional Hash Tables. In *Workshop on Distributed Data and Structures* (2006).

[13] FAL LABS. Tokyo Cabinet: a modern implementation of DBM. http://fallabs.com/tokyocabinet/.

[14] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI* (2013).

[15] FITZPATRICK, B. Distributed Caching with Memcached. *LINUX Journal 124* (Aug 2004).

[16] FOTAKIS, D., PAGH, R., SANDERS, P., AND SPIRAKIS, P. Space Efficient Hash Tables With Worst Case Constant Access Time. In *STACS* (2003).

[17] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch Hashing. In *DISC* (2008).

[18] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008).

[19] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating phase change memory for enterprise storage systems: a study of caching and tiering approaches. In *FAST* (2014).

[20] KIRSCH, A., AND MITZENMACHER, M. The Power of One Move: Hashing Schemes for Hardware. *Transactions on Networking 18*, 6 (Dec 2010).

[21] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal of Computing 39*, 4 (Dec. 2009).

[22] KNUTH, D. *The Art of Computer Programming, Volume 3: (2nd Edition) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[23] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[24] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *EuroSys* (2014).

[25] LIM, H., FAN, B., ANDERSEN, D., AND KAMINSKY, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *SOSP* (2011).

[26] MERRIMAN, D. A., AND O'CONNOR, K. J. Method of delivery, targeting, and measuring advertising over networks, Sept. 7 1999. US Patent 5,948,061.

[27] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling memcache at facebook. In *NSDI* (2013).

[28] PAGH, R., AND RODLER, F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (May 2004).

[29] PANIGRAHY, R. Efficient Hashing with Lookups in Two Memory Accesses. In *SODA* (2005).

[30] QURESHI, M. K., KARIDIS, J., FRANCESCHINI, M., SRINIVASAN, V., LASTRAS, L., AND ABALI, B. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *ISCA* (2009).

[31] ROSS, K. A. Efficient Hash Probes on Modern Processors. In *ICDE* (2007).

[32] SPOTIFY INC. New open source key-value store: sparkey. http://labs.spotify.com/2013/09/03/sparkey/.

[33] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST* (2011).

[34] VIGLAS, S. D. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment 7*, 5 (2014).

[35] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS* (2011).

[36] ZUKOWSKI, M., HÉMAN, S., AND BONCZ, P. Architecture-Conscious hashing. In *Workshop on Data Management on New Hardware (Damon)* (2006).