Dash: Scalable Hashing on Persistent Memory

Baotong Lu[1*]   Xiangpeng Hao[2]   Tianzheng Wang[2]   Eric Lo[1]

[1]The Chinese University of Hong Kong   [2]Simon Fraser University

{btlu, ericlo}@cse.cuhk.edu.hk   {xha62, tzwang}@sfu.ca

ABSTRACT

Byte-addressable persistent memory (PM) brings hash tables the potential of low latency, cheap persistence and instant recovery. The recent advent of Intel Optane DC Persistent Memory Modules (DCPMM) further accelerates this trend. Many new hash table designs have been proposed, but most of them were based on emulation and perform sub-optimally on real PM. They were also piece-wise and partial solutions that side-step many important properties, in particular good scalability, high load factor and instant recovery.

We present Dash, a holistic approach to building dynamic and scalable hash tables on real PM hardware with all the aforementioned properties. Based on Dash, we adapted two popular dynamic hashing schemes (extendible hashing and linear hashing). On a 24-core machine with Intel Optane DCPMM, we show that compared to state-of-the-art, Dash-enabled hash tables can achieve up to ∼3.9× higher performance with up to over 90% load factor and an instant recovery time of 57ms regardless of data size.

1. INTRODUCTION

Dynamic hash tables that can grow and shrink as needed at runtime are a fundamental building block of many data-intensive systems, such as database systems [11, 15, 26, 34, 41, 46] and key-value stores [5, 13, 16, 24, 37, 48, 62]. Persistent memory (PM) such as 3D XPoint [9] and memristor [53] promises byte-addressability, persistence, high capacity, low cost and high performance, all on the memory bus. These features make PM very attractive for building dynamic hash tables that persist and operate directly on PM, with high performance and instant recovery. The recent release of Intel Optane DC Persistent Memory Module (DCPMM) brings this vision closer to reality. Since PM exhibits several distinct properties (e.g., asymmetric read/write speeds and higher latency); blindly applying prior disk or DRAM based approaches [12, 29, 36] would not reap its full benefits, necessitating a departure from conventional designs.

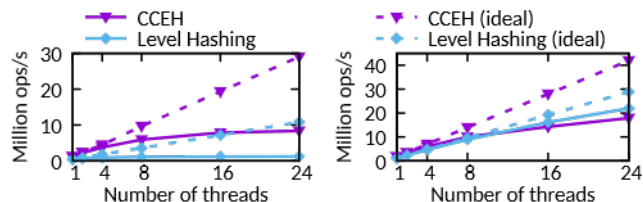*Work partially performed while at Simon Fraser University.

Figure 1: Throughput of state-of-the-art PM hashing (CCEH [38] and Level Hashing [68]) for insert (left) and search (right) operations on Optane DCPMM. Neither matches the expected scalability.

1.1 Hashing on PM: Not What You Assumed!

There have been a new breed of hash tables specifically designed for PM [10, 38, 39, 51, 67, 68] based on DRAM emulation, before actual PM was available. Their main focus is to reduce cacheline flushes and PM writes for scalable performance. But when they are deployed on real Optane DCPMM, we find (1) scalability is still a major issue, and (2) desirable properties are often traded off.

Figure 1 shows the throughput of two state-of-the-art PM hash tables [38, 68] under insert (left) and search (right) operations, on a 24-core server with Optane DCPMM running workloads under uniform key distribution (details in Section 6). As core count increases, neither scheme scales for inserts, nor even read-only search operations. Corroborating with recent work [31, 56], we find the main culprit is Optane DCPMM's limited bandwidth which is ∼3–14× lower than DRAM's [22]. Although the server is fully populated to provide the maximum possible bandwidth, excessive PM accesses can still easily saturate the system and prevent the system from scaling. We describe two major sources of excessive PM accesses that were not given enough attention before, followed by a discussion of important but missing functionality in prior work.

Excessive PM Reads. Much prior work focused on reducing writes to PM, however, we note that it is also imperative to reduce PM reads; yet many existing solutions reduce PM writes by incurring more PM reads. Different from the device-level behavior (PM reads being faster than writes), *end-to-end* write latency (i.e., the entire data path including CPU caches and write buffers in the memory controller) is often lower than reads [56, 63]. The reason is while PM writes can leverage write buffers, PM reads mostly touch the PM media due to hash table's inherent random access patterns. In particular, existence checks in record probing constitute a large proportion of such PM reads: to find out if a key exists, one or multiple buckets (e.g., with linear probing) have to be searched, incurring many cache misses and PM reads when comparing keys.

Heavyweight Concurrency Control. Most prior work sidestepped the impact of concurrency control. Bucket-level locking

has been widely used [38, 68], but it incurs additional PM writes to acquire/release read locks, further pushing bandwidth consumption towards the limit. Lock-free designs [51] can avoid PM writes for read-only probing operations, but are notoriously hard to get right, more so in PM for safe persistence [59].

Neither record probing nor concurrency control typically prevents a well-designed hash table to scale on DRAM. However, on PM they can easily exhaust PM's limited bandwidth. These issues call for new designs that can reduce unnecessary PM reads during probing and lightweight concurrency control that further reduces PM writes.

Missing Functionality. We observe in prior designs, necessary functionality is often traded off for performance (though scalability is still an issue on real PM). (1) Indexes could occupy more than 50% of memory capacity [66], so it is critical to improve load factor (records stored vs. hash table capacity). Yet high load factor is often sacrificed by organizing buckets using larger segments in exchange for smaller directories (fewer cache misses) [38]. As we describe later, this in turn can trigger more pre-mature splits and incur even more PM accesses, impacting performance. (2) Variable-length keys are widely used in reality, but prior approaches rarely discuss how to efficiently support it. (3) Instant recovery is a unique, desirable feature that could be provided by PM, but is often omitted in prior work which requires a linear scan of the metadata whose size scales with data size. (4) Prior designs also often side-step the PM programming issues (e.g., PM allocation), which impact the proposed solution's scalability and adoption in reality.

1.2 Dash

We present *Dash*, a holistic approach to dynamic and scalable hashing on real PM without trading off desirable properties. Dash uses a combination of new and existing techniques that are carefully engineered to achieve this goal. ❶ We adopt fingerprinting [43] that was used in PM tree structures to avoid unnecessary PM reads during record probing. The idea is to generate fingerprints (one-byte hashes) of keys and place them compactly to summarize the possible existence of keys. This allows a thread to tell if a key possibly exists by scanning the fingerprints which are much smaller than the actual keys. ❷ Instead of traditional bucket-level locking, Dash uses an optimistic, lightweight flavor of it that relies on verification to detect conflicts, rather than (expensive) shared locks. This allows Dash to avoid PM writes for search operations. With fingerprinting and optimistic concurrency, Dash avoids both unnecessary reads and writes, saving PM bandwidth and allowing Dash to scale well. ❸ Dash retains desirable properties. We propose a new load balancing strategy to postpone segment splits with improved space utilization. To support instant recovery, we limit the amount of work to be done upon recovery to be small (reading and possibly writing a one-byte counter), and amortize recovery work to runtime. Compared to prior work that handles PM programming issues in ad hoc ways, Dash uses PM programming models (PMDK [20], one of the most popular PM libraries) to systematically handle crash consistency, PM allocation and achieve instant recovery.

Although these techniques are not all new, Dash is the first to integrate them for building hash tables that scale without sacrificing features on real PM. Techniques in Dash can be applied to various static and dynamic hashing schemes. Compared to static hashing, dynamic hashing can adjust hash table size on demand without full-table rehashing which may block concurrent queries and significantly limit performance. In this paper, we focus on dynamic hashing and apply Dash to two classic approaches: extendible hashing [12, 38] and linear hashing [29, 36]. They are both widely used in database and storage systems, such as Oracle ZFS [40], IBM GPFS [49], Berkeley DB [3] and SQL Server Hekaton [32].

Evaluation using a 24-core Intel Xeon Scalable CPU and 1.5TB of Optane DCPMM shows that Dash can deliver high performance, good scalability, high space utilization and instant recovery with a constant recovery time of 57ms. Compared to the aforementioned state-of-the-art [38, 68], Dash achieves up to ∼3.9× better performance on realistic workloads, and up to over 90% load factor with high space utilization and the ability to instantly recover.

1.3 Contributions and Paper Organization

We make four contributions. First, we identify the mismatch between existing and desirable PM hashing approaches, and highlight the new challenges. Second, we propose Dash, a holistic approach to building scalable hash tables on real PM. Dash consists of a set of useful and general building blocks applicable to different hash table designs. Third, we present Dash-enabled extendible hashing and linear hashing, two classic and widely used dynamic hashing schemes. Finally, we provide a comprehensive empirical evaluation of Dash and existing PM hash tables to pinpoint and validate the important design decisions. Our implementation is open-source at: https://github.com/baotonglu/dash.

In the rest of the paper, we give necessary background in Section 2. Sections 3–5 present our design principles and Dash-enabled extendible hashing and linear hashing. Section 6 evaluates Dash. We discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND

We first give necessary background on PM (Optane DCPMM) and dynamic hashing, then discuss issues in prior PM hash tables.

2.1 Intel Optane DC Persistent Memory

Hardware. We target Optane DCPMMs (in DIMM form factor). In addition to byte-addressability and persistence, DCPMM offers high capacity (128/256/512GB per DIMM) at a price lower than DRAM's. It supports modes: *Memory* and *AppDirect*. The former presents capacious but slower volatile memory. DRAM is used as a cache to hide PM's higher latency, with hardware-controlled caching policy. The AppDirect mode allows software to explicitly access DRAM and PM with persistence in PM, without implicit caching. Applications need to make judicious use of DRAM and PM. Similar to other work [38, 39, 51, 68], we leverage the AppDirect mode, as it provides more flexibility and persistence guarantees.

System Architecture. The current generation of DCPMM requires the system be equipped with DRAM to function properly. We also assume such setup with PM and DRAM, both of which are behind multiple levels of *volatile* CPU caches. Data is not guaranteed to be persisted in PM until a cacheline flush instruction (CLFLUSH, CLFLUSHOPT or CLWB) [21] is executed or other events that implicitly cause cacheline flush occur. Writes to PM may also be reordered, requiring fences to avoid undesirable reordering. The application (hash table in our case) must explicitly issue fences and cacheline flushes to ensure correctness. CLFLUSH and CLFLUSHOPT will evict the cacheline that is being flushed, while CLWB does not (thus may give better performance). After a cacheline of data is flushed, it will reach the asynchronous DRAM refresh (ADR) domain which includes a write buffer and a write pending queue with persistence guarantees upon power failure. Once data is in the ADR domain, it is considered as persisted. Although DCPMM supports 8-byte atomic writes, internally it uses the 256-byte blocks. But software should not (be hardcoded to) depend on this as it is an internal parameter of the hardware which may change in future generations.

Performance Characteristics. At the device level, as many previous studies have shown, PM exhibits asymmetric read and write latency, with writes being slower. It exhibits ∼300ns read latency,

~4× longer than DRAM's. More recent studies [56, 63], however revealed that on Optane DCPMM, read latency *as seen by the software* is often higher than write latency. This is attributed to the fact that writes (`store` instructions) commit once the data reaches the ADR domain at the memory controller rather than when reaching DCPMM media. On the contrary, a read operation often needs to touch the actual media unless the data being accessed is cache-resident (which is rare especially in data structures with inherent randomness, e.g., hash tables). Tests also showed that the bandwidth of DCPMM depends on many factors of the workload. In general, compared to DRAM, it exhibits ~3×/~8× slower sequential/random read bandwidth. The numbers for sequential/random write are ~11×/~14×. Notably, the performance of small random stores is severely limited and non-scalable [63], which, however, is the inherent access pattern in hash tables. These properties exhibit a stark contrast to prior estimates about PM performance [45, 55, 58], and lead to significantly lower performance of many prior designs on DCPMM than originally reported. Thus, it is important to reduce *both* PM reads and writes for higher performance. More details on raw DCPMM device performance can be found elsewhere [22]; we focus on the end-to-end performance of hash tables on PM.

2.2 Dynamic Hashing

Now we give an overview of extendible hashing [12] and linear hashing [29, 36]. We focus on their memory-friendly versions which PM-adapted hash tables were based upon. Dash can also be applied to other approaches which we defer to future work.

Extendible Hashing. The crux of extendible hashing is its use of a directory to index buckets so that they can be added and removed dynamically at runtime. When a bucket is full, it is split into two new buckets with keys redistributed. The directory may get expanded (doubled) if there is not enough space to store pointers to the new bucket. Figure 2(a) shows an example with four buckets, each of which is pointed to by a directory entry; a bucket can store up to four records (key-value pairs). In the figure, indices of directory entries are shown in binary. The two least significant bits (LSBs) of the hash value are used to select a bucket; we call the number of suffix bits being used here the *global depth*. The hash table can have at most $2^{global\ depth}$ directory entries (buckets). A search operation follows the pointer in the corresponding directory entry to probe the bucket. Each bucket also has a *local depth*. In Figure 2(a), the local depth of each bucket is 2, same as the global depth. Suppose we want to insert key 30 that is hashed to bucket 01_2, which is full and needs to be split to accommodate the new key.[1] Splitting the bucket will require more directory entries. In extendible hashing, the directory always grows by doubling its current size. The result is shown in Figure 2(b). Here, bucket 01_2 in Figure 2(a) is split into two new buckets (001_2 and 101_2), one occupies the original directory entry, and the other occupies the second entry in the newly added half of the directory. Other new entries still point to their corresponding original buckets. Search operations will use three bits to determine the directory entry index (global depth now is three).

After a bucket is split, we increment its local depth by one, and update the new bucket's local depth to be the same (3 in our example). The other unsplit buckets' local depth remains 2. This allows us to determine whether a directory doubling is needed: if a bucket whose local depth equals the global depth is split (e.g., bucket 001_2 or 101_2), then the directory needs to be doubled to accommodate the new bucket. Otherwise (local depth < global depth), the directory should have $2^{global\ depth - local\ depth}$ directory entries pointing to that bucket, which can be used to accommodate the new bucket.

---

[1]Choosing a proper hash function that evenly distributes keys to all buckets is an important but orthogonal problem to our work.
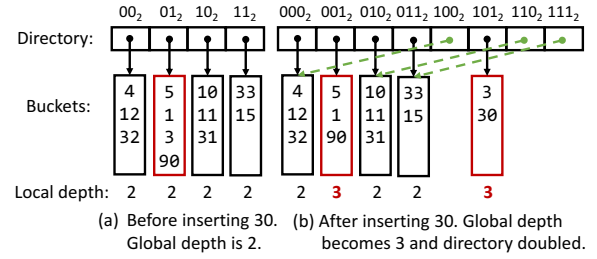

(a) Before inserting 30. Global depth is 2. (b) After inserting 30. Global depth becomes 3 and directory doubled.

Figure 2: An example of extendible hashing. **(a)** The hash table is full. **(b)** The local depth of unsplit buckets is 2. Splitting buckets with local depth < global depth will not double the directory.

For instance, if bucket 000_2 needs to be split, directory entry 100_2 (pointing to bucket 000_2) can be updated to point to the new bucket.

Linear Hashing. In-memory linear hashing takes a similar approach to organizing buckets using a directory with entries pointing to individual buckets [29]. The main difference compared to extendible hashing is that in linear hashing, the bucket to be split is chosen "linearly." That is, it keeps a pointer (page ID or address) to the bucket to be split next and only that bucket would be split in each round, and advances the pointer to the next bucket when the split of the current bucket is finished. Therefore, the bucket being split is not necessarily the same as the bucket that is full as a result of inserts, and eventually the overflowed bucket will be split and have its keys redistributed. If a bucket is full and an insert is requested to it, more overflow buckets will be created and chained together with the original, full bucket. For correct addressing and lookup, linear hashing uses a group of hash functions $h_1...h_n$, where h_n covers twice the range of h_{n-1}. For buckets that are already split, h_n is used so we can address buckets in the new hash table capacity range, and for the other unsplit buckets we use h_{n-1} to find the desired bucket. After all buckets are split (a *round* of splitting has finished), the hash table's capacity will be doubled; the pointer to the next-to-be-split bucket is reset to the first bucket for the next round of splitting.

Determining when to trigger a split is an important problem in linear hashing. A typical approach is to monitor and keep the load factor bounded [29]. The choice of a proper splitting strategy may also vary across workloads, and is orthogonal to the design of Dash.

2.3 Dynamic Hashing on PM

Now we discuss how dynamic hashing is adapted to work on PM. We focus on extendible hashing and start with CCEH [38], a recent representative design; Section 5 covers linear hashing on PM.

To reduce PM accesses, CCEH groups buckets into *segments*, similar to in-memory linear hashing [29]. Each directory entry then points to a segment which consists of a fixed number of buckets indexed by additional bits in the hash values. By combining multiple buckets into a larger segment, the directory can become significantly smaller as fewer bits are needed to address segments, making it more likely to be cached entirely by the CPU, which helps reducing access to PM. Note that split now happens at the segment (instead of bucket) level. A segment is split once any bucket in it is full, even if the other buckets in the segment still have free slots, which results in low load factor and more PM accesses. To reduce such pre-mature splits, linear probing can be used to allow a record to be inserted into a neighbor bucket. However, this improves load factor at the cost of more cache misses and PM accesses. Thus, most approaches bound probing distance to a fixed number, e.g., CCEH probes no more than four cachelines. However, our evaluation (Section 6) shows that linear probing alone is not enough in achieving high load factor.

Another important aspect of dynamic PM hashing is to ensure failure atomicity, particularly during segment split which is a three-step process: (1) allocate a new segment in PM, (2) rehash records from the old segment to the new segment and (3) register the new segment in the directory and update local depth. Existing approaches such as CCEH only focused on step 3, side-stepping PM management issues surrounding steps 1–2. If the system crashes during step 1 or 2, we need to guarantee the new segment is reclaimed upon restart to avoid *permanent* memory leaks. In Sections 4 and 6.1, we describe Dash's solution and a solution for existing approaches.

3. DESIGN PRINCIPLES

The aforementioned issues and performance characteristics of Optane DCPMM lead to the following design principles of Dash:

- **Avoid both Unnecessary PM Reads and Writes.** Probing performance impacts not only search operations, but also all the other operations. Therefore, in addition to reducing PM writes, Dash must also remove unnecessary PM reads to conserve bandwidth and alleviate the impact of high end-to-end read latency.

- **Lightweight Concurrency.** Dash should scale well on multicore machines with persistence guarantees. Given the limited bandwidth, concurrency control must be lightweight to not incur much overhead (i.e., avoid PM writes for search operations, such as read locks). Ideally, it should also be relatively easy to implement.

- **Full Functionality.** Dash must not sacrifice or trade off important features that make a hash table useful in practice. In particular, it needs to support near-instantaneous recovery and variable-length keys and achieve high space utilization.

4. Dash FOR EXTENDIBLE HASHING

Based on the principles in Section 3, we describe Dash in the context of Dash-Extendible Hashing (Dash-EH). We discuss how Dash applies to linear hashing in Section 5.

4.1 Overview

Similar to prior approaches [38, 65], Dash-EH uses segmentation. As shown in Figure 3, each directory entry points to a segment which consists of a fixed number of normal buckets and stash buckets for overflow records from normal buckets which did not have enough space for the inserts. The lock, version number and clean marker are for concurrency control and recovery, which we describe later.

Figure 4 shows the internals of a bucket. We place the metadata used for bucket probing on the first 32 bytes, followed by multiple 16-byte record (key-value pair) slots. The first 8 bytes in each slot store the key (or a pointer to it for keys longer than 8 bytes). The remaining 8 bytes store the payload which is opaque to Dash; it can be an inlined value or a pointer, depending on the application's need. The size of a bucket is adjustable. In our current implementation it is set to 256-byte (block size of Optane DCPMM [22]) for better locality. This allows us to store 14 records (16-byte each) per bucket.

The 32-byte metadata includes key data structures for Dash-EH to handle hash table operations and realize the design principles. It starts with a 4-byte version lock for optimistic concurrency control (Section 4.4). A 4-bit `counter` records the number of records stored in the bucket. The `allocation` bitmap reserves one bit per slot, to indicate whether the corresponding slot stores a valid record. The `membership` bitmap is reserved for bucket load balancing which we describe later (Section 4.3). What follows are structures such as fingerprints and counters to accelerate probing and improve load factor. Most unnecessary probings are avoided by scanning the fingerprints area.
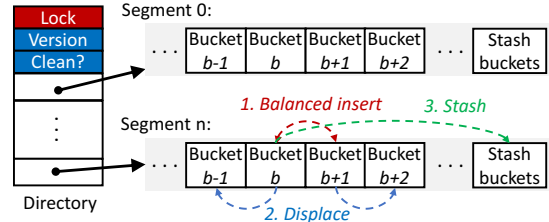

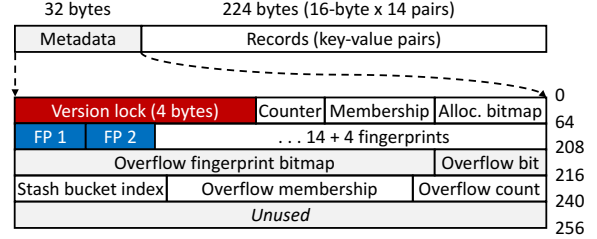**Figure 3:** Overall architecture of Dash-EH.

Figure 4: Dash-EH bucket layout. The first 32 bytes are dedicated to metadata that optimizes probing and load factor, followed by records. Normal and stash buckets share the same layout.

4.2 Fingerprinting

Bucket probing (i.e., search in one bucket) is a basic operation needed by all the operations supported by a hash table (search, insert and delete) to check for key existence. Searching a bucket typically requires a linear scan of the slots. This can incur lots of cache misses and is a major source of PM reads, especially so for long keys stored as pointers. It is a major reason for hash tables on PM to exhibit low performance. Moreover, such scans for negative search operations (i.e., when the target key does not exist) are completely unnecessary.

We employ fingerprinting [43] to reduce unnecessary scans. It was used by trees to reduce PM accesses with an amortized number of key loads of one. We adopt it in hash tables to reduce cache misses and accelerate probing. Fingerprints are one-byte hashes of keys for predicting whether a key possibly exists. We use the least significant byte of the key's hash value. To probe for a key, the probing thread first checks whether any fingerprint matches the search key's fingerprint. It then only accesses slots with matching fingerprints, skipping all the other slots. If there is no match, the key is definitely not present in the bucket. This process can be further accelerated with SIMD instructions [21].

Fingerprinting particularly benefits negative search (where the search key does not exist) and uniqueness checks for inserts. It also allows Dash to use larger buckets to tolerate more collisions and improve load factor, without incurring many cache misses: most unnecessary probes are avoided by fingerprints. This design contrasts with many prior designs that trade load factor for performance by having small buckets of 1–2 cachelines [10, 38, 68].

As Figure 4 shows, each bucket contains 14 slots, but 18 fingerprints (bits 64–208); 14 are for slots in the bucket, and the other four represent keys placed in a stash bucket but were originally hashed into the current bucket. They can allow early avoidance of access to stash buckets, saving PM bandwidth. We describe details next as part of the bucket load balancing strategy that improves load factor.

4.3 Bucket Load Balancing

Segmentation reduces cache misses on the directory (by reducing its size). However, as we describe in Sections 2.3 and 6, this is at the cost of load factor: in a naive implementation the entire segment

needs to be split if any bucket is full, yet other buckets in the segment may still have much free space. We observe that the key reason is load imbalance caused by the (inflexible) way buckets are selected for inserting new records, i.e., a key is only mapped to a single bucket. Dash uses a combination of techniques for new inserts to balance loads among buckets while limiting PM reads needed. Algorithm 1 shows how the insert operation works in Dash-EH at a high level, with three key techniques described below.

Balanced Insert. To insert a record whose key is hashed into bucket b ($hash(key) = b$), Dash probes both bucket b and $b+1$ and inserts the record into the bucket that is less full (Figure 3 step 1). Lines 17–21 in Algorithm 1 show the idea; we discuss how a record is inserted into a bucket later. The rationale behind is to improve load factor by amortizing the load of hot buckets while limiting PM accesses (at most two buckets). Compared to balanced insert, linear probing allows a record to be inserted into bucket $b+n$ where $n > 1$ if buckets $b...b+n-1$ are full. Probing multiple buckets may degrade performance by imposing more PM reads and cache misses. It is also hard to tune the number of buckets to probe.

Displacement. If both the target bucket b and probing bucket $b+1$ are full, Dash-EH tries to displace (move) a record from bucket b or $b+1$ to make room for the new record (Algorithm 1 lines 23–26). With balanced insert, a record in bucket $n+1$ can be moved to $n+2$ if (1) it could be inserted to either bucket (i.e., $n+2$ is the probing bucket of the record being moved), *and* (2) bucket $n+2$ has a free slot. Thus, for a record with $hash(key) = b$ and both b and $b+1$ are full, we first try to find a record in $b+1$ whose $hash(key) = b+1$ and move it to $b+2$. If such a record does not exist, we repeat for bucket b but move a record with $hash(key) = b-1$ (the target bucket). In essence, displacement follows a similar strategy to balanced insert, but is for existing records.

We use a per-bucket `membership` bitmap (Figure 3) to accelerate displacement. If a bit is set, then the corresponding key was not originally hashed into this bucket; it was placed here because of balanced insert or displacement. When checking for bucket b ($b+1$), a record whose membership bit is set (unset) can be displaced. Dash then only needs to scan the bitmap to pick directly a record to move, without having to examine the actual keys. This reduces unnecessary PM accesses, and is especially beneficial for variable length keys which are not inlined but represented by pointers.

Stashing. If the record cannot be inserted into bucket b or $b+1$ after balanced insert and displacement, stashing will be the last resort before segment split has to happen. In Figure 3, a tunable number of stash buckets follow the normal buckets in each segment. If a record cannot be inserted into its target bucket nor the probing bucket, we insert the record to a stash bucket; we call these records *overflow records*. Stash buckets use the same layout as that of normal buckets; probing of a stash bucket follows the same procedure as probing a normal bucket (see Section 4.2). While stashing can be effective in improving load factor, it could incur non-trivial overhead: the more stash buckets are used, the more CPU cycles and PM reads will be needed to probe them. This is especially undesirable for negative search and uniqueness check in insert operations, since both need to probe all stash buckets, despite it may be completely unnecessary.

To solve this problem, we try to set up record metadata including fingerprints in a normal bucket and only refer actual record access to the stash bucket. As Figure 4 shows, four additional fingerprints per bucket are reserved for overflow records stored in stash buckets. A 4-bit `overflow fingerprint` bitmap records whether the corresponding fingerprint slot is occupied. Another `overflow bit` indicates whether the bucket has overflowed any record to a stash bucket. Algorithm 1 (lines 27–29) shows this process. Similar to inserting records into a normal bucket, the overflow record's fin-

---

Algorithm 1 Dash-EH insert algorithm with bucket load balancing.

```
 1 def dash_eh_insert(key, value):
     h = hash(key)
 3 retry:
     # Obtain references and lock buckets
 5   [target_seg] = get_segment(h)
     [target_bucket, probing_bucket] = target_seg.bk(h)
 7   Lock target_bucket and probing_bucket

 9   # Verify the correctness of the segment reference
     [verify_seg] = get_segment(h)
11   if verify_seg is not target_seg:
       Unlock and goto retry
13
     if key exists in either bucket or the stash:
15     Unlock and return Result::KeyExists

17   if target_bucket or probing_bucket is not full:
       if target_bucket.count <= probing_bucket.count:
19       target_bucket.insert(key, value, h)
       else
21       probing_bucket.insert(key, value, h)
     else
23     # Try displacement (possibly stashing)
       bucket = displace(target_bucket, probing_bucket)
25     if bucket is not NULL:
         bucket.insert(key, value, h)
27     elif stash_bucket.insert(key, value, h):
         target.overflow = true
29       Set overflow fingerprint bitmap and fingerprint
       else # Stashing failed, have to split
31       split_segment(h)
         goto retry
33   Unlock target_bucket and probing_bucket
     return Result::Inserted
```

---

gerprint also allows one more bucket probing, using the `overflow membership` bitmap to indicate whether the overflow fingerprints originally belong to this bucket. The 2-bit `stash bucket index` per overflow record indicates which one of the four stash buckets the record is inserted into for faster lookup. If the overflow fingerprint cannot be inserted into neither the target nor the probing bucket, we increment `overflow count` in the target bucket. Once the counter becomes positive, a probing thread will have to check the stash area to ensure that a key does or does not exist. Thus, it is desirable to reserve enough slots of overflow fingerprints in each bucket so that the overflow counter is rarely positive. As Section 6 shows, using 2–4 stash buckets per segment can improve load factor to over 90% without imposing significant overhead.

4.4 Optimistic Concurrency

Dash employs optimistic locking, an optimistic flavor of bucket-level locking inspired by optimistic concurrency control [27, 54]. Insert operations will follow traditional bucket-level locking to lock the affected buckets. Search operations are allowed to proceed without holding any locks (thus avoiding writes to PM) but need to verify the read record. For this to work, in Dash the lock consists of (1) a single bit that serves the role of "the lock" and (2) a version number for detecting conflicts (not to be confused with the version number in Figure 3 for instant recovery). As line 7 in Algorithm 1 shows, the inserting thread will acquire bucket-level locks for the target and probing buckets. This is done by atomically setting the lock bit in each bucket by trying the `compare-and-swap` (CAS) instruction [21] until success. Then the thread enters the critical section and continues its operations. After the insert is done, the thread

releases the lock by (1) resetting the lock bit and (2) incrementing the version number by one, in one step using an atomic write.

To probe a bucket for a key, Dash first takes a snapshot of the lock word and checks whether the lock is being held by a concurrent writer (the lock bit is set). If so, it waits until the lock is released and repeats. Then it is allowed to read the bucket *without* holding any lock. Upon finishing its operations, the reader thread will read the lock word again to verify the version number did not change, and if so, it retries the entire operation as the record might not be valid as a concurrent write might have modified it. This lock-free read design requires segment deallocation (due to merge) happen only after no readers are (and will be) using the segment. We use epoch-based reclamation [14] to achieve this without incurring much overhead.

Dash does not use segment-level locks, saving PM access in the segment level. As a result, structural modification operations (SMOs, such as segment split) need to lock all the buckets in each segment. Directory doubling/halving is handled similarly: the directory lock is only held when the directory is being doubled or halved. For other operations on the directory (e.g., updating a directory entry to point to a new segment), no lock is taken. Instead, they are treated as search operations without taking the directory lock. This is safe because we guarantee isolation in the segment level: an inserting thread must first acquire locks to protect the affected buckets. "Real" probings (search/insert) proceed without reading the directory lock but again need to verify that they entered the right segment by re-reading the directory to test whether these two read results match; if not, the thread aborts and retries the entire operation.

4.5 Support for Variable-Length Keys

Dash stores pointers to variable-length keys, which is a common approach [2, 38, 43, 68]. A knob is provided to switch between the inline (fixed-length keys up to 8 bytes) and pointer modes. Though dereferencing pointers may incur extra overhead, fingerprinting largely alleviates this problem. For negative search where the target key does not exist, no fingerprint will match and so key probing will not happen at all. For positive search, as we have discussed in Section 4.2, the amortized number of key load (therefore the number cache misses caused by following the key pointer) is one [43].

4.6 Record Operations

Now we present details on how Dash-EH performs insert, search and delete operations on PM with persistence guarantees.

Insert. Section 4.3 presented the high-level steps for insert; here we focus on the bucket-level. As the `bucket::insert` function in Algorithm 2 shows, we first write and persist the new record (lines 3-4), and then set up the metadata (fingerprint, allocation bitmap, counter and membership, lines 6–12). Note that the allocation bitmap, membership bitmap and counter are in one word; they are updated in one atomic write. The CLWB and fence are then issued (line 16) to persist all the metadata. Once the corresponding bit in the bitmap is set, the record is visible to other threads. If a crash happens before the bitmap is persisted, the new record is regarded as invalid; otherwise, the record is successfully inserted. This allows us to avoid expensive logging while maintaining consistency.

Displacing a record needs two steps: (1) inserting it into the new bucket and (2) deleting it from the original bucket. As the `displace` function in Algorithm 2 shows, step 2 is done by resetting the corresponding bit in the allocation bitmap without moving data. In case a crash happens before step 2 finishes, a record will appear in both buckets. This necessitates a duplicate detection mechanism upon recovery, which is amortized over runtime (see Section 4.8).

If the insert has to happen in a stash bucket, we set the overflow metadata in the normal bucket. This cannot be done atomically with

---

Algorithm 2 Bucket insert and displacement in Dash-EH.

```
   def bucket::insert(key, value, h):
2    slot = slots[slot_id = get_free_slot()]
     slot.assign(key, value)
4    CLWB+FENCE(slot) # Persist the record first

6    fingerprints[slot_id] = LSB_byte(h)
     # Since following metadata are in the same word,
8    # their updates are done in one store operation
     alloc_bitmap.set(slot_id)
10   ++counter
     if bucket is probing bucket:
12     membership.set(slot_id)

14   # Persist all metadata updates in one flush
     # (same cacheline, no reordering on x86)
16   CLWB+FENCE(alloc_bitmap, membership, counter, FP)

18 def displace(target = b, prob = b+1):
     # Try to move a record from bucket b+1 to b+2
20   slot_id = prob.get_unset_LSB(membership)
     if slot_id is not Invalid and (b+2) is not full:
22     (prob+1).insert(prob.slots[slot_id])
       # Mark deletion for the moved record, decrement
24     # the counter (done in one store operation)
       prob.alloc_bitmap &= ~(1 << slot_id)
26     prob.counter--
       CLWB+FENCE(prob.alloc_bitmap, counter)
28     return prob
     else
30     # Try to move a record from bucket b to b-1
       slot_id = target.get_set_LSB(membership)
32     if slot_id is not Invalid and (b-1) is not full:
         (target-1).insert(target.slots[slot_id])
34       target.alloc_bitmap &= ~(1 << slot_id)
         target.membership &= ~(1 << slot_id)
36       target.counter--
         CLWB+FENCE(target.alloc_bitmap, counter)
38       return target
       else
40       return NULL
```

8-byte writes and may need a (complex) protocol for crash consistency. We note that the overflow metadata is an optimization and does not influence correctness: records can still be found correctly even without it. So we do not explicitly persist it and rely on the lazy recovery mechanism to build it up gradually (described later).

Search. With balanced insert and displacement, a record could be inserted into its target bucket b where $b = hash(key)$ or its probing bucket $b + 1$. A search operation then has to check both if the record is not found in b. As Algorithm 3 shows, the probing thread first reads the directory to obtain a reference to the corresponding segment and buckets (lines 4–5). It then takes a snapshot of the version number of both buckets (lines 6–7) for verification later. We verify at line 10 that the segment did not change (i.e., the directory entry still points to it) and retry if needed. Once segment check passed, we check whether the target/probing buckets are being modified (i.e., locked) at lines 14–15. If not, we continue to search the target and probing buckets (lines 17–27) using the `bucket::search` function (not shown). Note that we need to verify the lock version did not change after `bucket::search` returns (lines 18 and 24).

If neither bucket contains the record, it might be stored in a stash bucket (lines 31–37). If `overflow_count > 0`, then we search the stash buckets as the overflow fingerprint area does not have enough space for all overflow records from the bucket. Otherwise, stash access is only needed if there is a matching fingerprint (lines 31–35).

Algorithm 3 Dash-EH search algorithm.

```
   def dash_eh_search(key):
 2 retry:
     # Get the segment and buckets
 4   [target_seg] = get_segment(h)
     [target_bucket, probing_bucket] = target_seg.bk(h)
 6   vt = target_bucket.version_lock
     vp = probing_bucket.version_lock
 8
     # Verify the correctness of the segment reference
10   [verify_seg] = get_segment(h)
     if verify_seg is not target_seg:
12     goto retry
14   if is_lock_set(vt) or is_lock_set(vp):
       goto retry
16
     result = target_bucket.search(key)
18   if vt is not target_bucket.version_lock:
       goto retry
20   if result is not NULL:
       return result
22
     result = probing_bucket.search(key)
24   if vp is not probing_bucket.version_lock:
       goto retry
26   if result is not NULL:
       return result
28
     # Determine whether to search in the stash buckets
30   # Note that version lock check is omitted below
     if probing bucket.overflow_count is zero:
32     if key matches overflow fingerprints:
         search corresponding stash buckets and return
34     else
         return NULL
36   else
       Search by scanning the stash buckets and return
38   return NULL
```

Delete. To delete a record from a normal bucket, we reset the corresponding bit in the allocation bitmap, decrement the counter and persist these changes. Then the slot becomes available for future reuse. To delete a record from a stash bucket, in addition to clearing the bit in the allocation bitmap, we also clear the overflow fingerprint in the target bucket which this record overflowed from if it exists; otherwise we only decrement the target bucket's overflow counter.

4.7 Structural Modification Operations

When a thread has exhausted all the options to insert a record into a bucket, it triggers a segment split and possibly expansion of the directory. Conversely, when the load factor drops below a threshold, segments can be merged to save space. At a high level, three steps are needed to split a segment S: (1) allocate a new segment N, (2) rehash keys in S and redistribute records in S and N, and (3) attach N to the directory and set the local depth of N and S. These operations cause the structure of the hash table to change and must be made crash consistency on PM while maintaining high performance.

For crash consistency, Dash-EH chains all segments using side links to the right neighbor. Each segment has a `state` variable to indicate whether the segment is in an SMO and whether it is the one being split or the new segment. An initial value of zero indicates the segment is not part of an SMO. Figure 5 shows an example. Note that as shown by the figure, Dash-EH uses the most significant bits (MSBs) of hash values to address and organize segments and buckets (i.e., the directory is indexed by the MSBs of hash values),
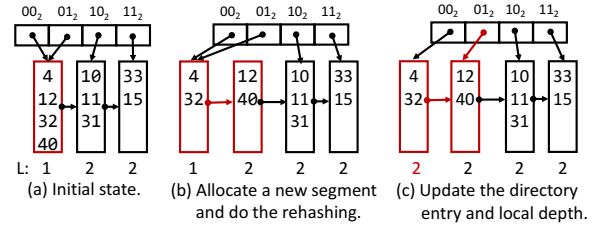

Figure 5: Segment split in Dash-EH; the global depth is 2.

similar to other recent work [38]. This is different from traditional extendible hashing described in Section 2.2 that uses LSBs of hash values to address buckets. Using LSBs was the choice in the disk era to reduce I/O: the directory can be doubled by simply copying the original directory and appending it to the directory file. On PM, such advantage is marginalized as to double a directory, one needs to allocate and persist a double-sized directory in PM anyway to keep the directory in a contiguous address space. Using MSBs also allows directory entries pointing to the same segment to be co-located, reducing cacheline flushes during splits [38].

To split a segment S, we first mark its `state` as SPLITTING and allocate a new segment N whose address is stored in the side link of S. N is then initialized to carry S's side link as its own. Its local depth is set to the local depth of S plus one. Then, we change N's state to NEW to indicate this new segment is part of a split SMO for recovery purposes (see Section 4.8). We rely on PM programming libraries (PMDK [20]) to atomically allocate *and* initialize the new segment; in case of a crash, the allocated PM block is guaranteed to be either owned by Dash or the allocator and will not be permanently leaked. After initialization, we finish up step 2 by redistributing records between N and S. Records moved from S to N are deleted in S after they are inserted into N. Note that the rehashing/redistributing process does not need to be done atomically: if a crash happens in the middle of rehashing, upon (lazy) recovery we redo the rehashing process with uniqueness check to avoid repeating work for records that were already inserted into N before the crash; we describe more details later in Section 4.8. Figure 5(b) shows the state of the hash table after step 2. Then the directory entry for N and the local depth of S are updated as shown in Figure 5(c). Similarly, these updates are conducted using an atomic PMDK transaction which may use any approach such as lightweight logging. Many other systems avoid the use of logging to maintain high performance, largely because of the frequent pre-mature splits. But split is much rarer in Dash thanks to bucket load balancing that gives high load factor (Section 4.3); this allows Dash-EH to employ logging-based PMDK transactions that abstracts away many details and eases implementation.

4.8 Instant Recovery

Dash provides truly instant recovery by requiring a constant amount of work (reading and possibly writing a one-byte counter), before the system is ready to accept user requests. We add a global version number V and a `clean` marker shown in Figure 3, and a per-segment version number. `clean` is a boolean that denotes whether the system was shutdown cleanly; V tells whether recovery (during runtime) is needed. Upon a clean shutdown, `clean` is set to true and persisted. Upon restart, if `clean` is true, we set `clean` to false and start to handle requests. Otherwise, we increment V by one and start to handle requests. For both clean shutdown and crash cases, "recovery" only involves reading `clean` and possibly bumping V. The actual recovery work is amortized over segment accesses.

To access a segment, the accessing thread first checks whether the segment version matches V. If not, the thread (1) recovers the

segment to a consistent state before doing its original operation (e.g., insert or search), and (2) sets the segment's version number to V so that future accesses can skip the recovery pass. With such lazy recovery approach, a segment is not recovered until it is accessed. Multiple threads may access a segment that needs to be recovered. We employ a segment-level lock that is only for recovery purpose, but a thread only tries to acquire the lock if it sees the segment's version number does not match V. Our current implementation uses one-byte version numbers. In case the version number wraps around and recovery is needed, we reset V to zero and set the version number of each segment to one. Since crash and repeated crashes are rare events, such wrap-around cases should be very rare.

Recovering a segment needs four steps: (1) clear bucket locks, (2) remove duplicate records, (3) rebuild overflow metadata, and (4) continue the ongoing SMO. Some locks might be in the locked state upon crash, so every lock in each bucket needs to be reset. Duplicate records are detected by checking the fingerprints in neighboring buckets. This is lightweight since the real key comparison is only needed if the fingerprints match. Overflow metadata in normal buckets also needs to be cleared and rebuilt based on the records in stash buckets as we do not guarantee their crash consistency for performance reasons. Finally, if a segment is in the SPLITTING state, the accessing thread will follow the segment's side link to test whether the neighbor segment is in the NEW state. If so, we restart the rehashing-redistribution process and finish the split. Otherwise, we reset the state variable which in effect rolls back the split.

5. Dash FOR LINEAR HASHING

We present Dash-LH, Dash-enabled linear hashing that uses the building blocks discussed previously (balanced insert, displacement, fingerprinting and optimistic concurrency). We do not repeat them here and focus on the design decisions specific to linear hashing.

5.1 Overview

Figure 6 (focus on segments 0-3 for now) shows the overall structure of Dash-LH. Similar to Dash-EH, Dash-LH also uses segmentation and splits at the segment level. However, we follow the linear hashing approach to always split the segment pointed to by the Next pointer, which is advanced after the segment is split. Since the segment to be split is not necessarily a full segment, it needs to be able to accommodate overflow records, e.g., using linked lists. However, linked list traversal would incur many cache misses, which is a huge penalty for PM hash tables. Instead, we leverage the stashing design in Dash and use an adjustable number of stash buckets. In addition to a fixed number of stash buckets (e.g., 2 stash buckets) in each segment, we store a linked list of stash buckets. A segment split is triggered whenever a stash bucket is allocated to accommodate overflow records. This contrasts with classic linear hashing which splits a bucket at a time which is vulnerable to long overflow chains under high insertion rate. Dash-LH uses larger split unit (segment) and chaining unit (stash bucket rather than individual records), reducing chain length (therefore pointer chasing and cache misses). The overflow metadata and fingerprints further helps alleviate the performance penalty brought by the need to search stash buckets. Overall, as we show in Section 6, Dash-LH can also achieve near-linear scalability on realistic workloads.

5.2 Hybrid Expansion

Similar to Dash-EH, it is also important to reduce directory size for better cache locality. Some designs use double expansion [17] which increases segment size exponentially: allocating a new segment doubles the number of buckets in the hash table. For example, the second segment allocated would be $1\times$ the size of the first
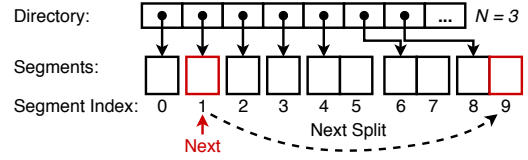

Figure 6: Overview of Dash-LH. Segments are organized in arrays.

segment, and the third segment would be $2\times$ larger than the first segment, and so on. The benefit is that directory size can become very small and be often fit even in the L1 cache. However, it also makes load factor drop by half whenever a new segment is allocated.

To reduce space waste, we postpone double expansion and expand the hash table by several fixed-size segments first, before triggering double expansion. We call the number of such fixed expansions the stride. Figure 6 shows an example (stride = 4). A directory entry can point to an array of segments; the first four entries point to one-segment arrays, the next four entries point to two-segment arrays, and so on. With a larger stride, the allocation of larger segment arrays will have less impact on load factor. The result is very small directory size that is typically L1-resident. Using 16KB segments, the first segment array will include 64 segments, with a stride of four, we can index TB-level data with a directory less than 1KB.

5.3 Concurrency

Since linear hashing expands in one direction, splits are essentially serialized by locking the Next pointer. To shorten the length of the critical section, we adopt the expansion strategy proposed by LHlf [65] where the expansion only atomically advances Next without actually splitting the segment. Then any thread that accesses a segment that should be split (denoted in the segment metadata area) will first conduct the actual split operation. As a result, multiple segments splits can execute in parallel by multiple threads. Before advancing the Next pointer, the accessing thread first probes the directory entry for the new segment to test whether the corresponding segment array is allocated. If not, it allocates the segment array and stores it in the directory entry. The performance of PM allocator therefore may impact overall performance, as we show in Section 6.

Dash-LH uses a variable N to compute the number of buckets of the base table. After each round of the split, Next is reset to zero and N is incremented to denote that the number of buckets is doubled. For consistency guarantees, we store N (32-bit) and Next (32-bit) in a 64-bit word which can be updated atomically.

6. EVALUATION

This section evaluates Dash and compares it with two other state-of-the-art PM hash tables, CCEH [38] and level hashing [68]. Specifically, through experiments we confirm the following:

- Dash-enabled hash tables (Dash-EH and Dash-LH) scale well on multicore servers with real Optane DCPMM;

- The bucket load balancing techniques allow Dash to achieve high load factor while maintaining high performance;

- Dash provides instant recovery with a minimal, constant amount of work needed upon restart, reducing service downtime.

6.1 Implementation

We implemented Dash-EH/LH using PMDK [20], which provides primitives for crash-safe PM management and synchronization. These primitives are essential for building PM data structures, but also introduce overhead. For example, PMDK allocator exhibits

scalability problems and is much slower than DRAM allocators [31]. Such overheads are ignored in previous emulation-based work, but are not negligible in reality. We take them into account in our evaluation. The other hash tables under comparison (CCEH [38] and level hashing [68]) were both proposed based on DRAM emulation. We ported them to run on Optane DCPMM using their original code[2] and PMDK. Like previous work [38], we optimize level hashing by co-locating all the locks in a small and continuous memory region (lock striping) [18] to reduce cache misses. Now we summarize the key implementation issues and our solutions.

Crash Consistency. Dash uses PMDK transactions for segment splits. This frees Dash from handling low-level details while guaranteeing safe and atomic allocations. We noticed a consistency issue in CCEH code where a power failure during segment split could leak PM. We fixed this problem using PMDK transaction. We also adapted CCEH and level hashing to use PMDK reader-writer locks that are automatically unlocked upon recovery.

Persistent Pointers. Both CCEH and level hashing assume standard 8-byte pointers based on DRAM emulation, while some systems use 16-byte pointers for PM [20, 25]. Long pointers break the memory layout and make atomic instructions hard to use. To use 8-byte pointers on PM, we extended PMDK to ensure that PM is mapped onto the same virtual address range across different runs (using `MAP_FIXED_NOREPLACE`[3] in mmap and setting `mmap_min_addr` in the kernel). All hash tables experimented here use this approach.

Garbage Collection. We implemented a general-purpose epoch-based PM reclamation mechanism for Dash. We also observed that the open-sourced implementation of CCEH allows threads to access the directory without acquiring any locks, which may allow access to freed memory (due to directory doubling or halving). We fixed this problem with the same epoch-based reclamation approach.

6.2 Experimental Setup

We run experiments on a server with a Intel Xeon Gold 6252 CPU clocked at 2.1GHz, 768GB of Optane DCPMM (6×128GB DIMMs on all six channels) in AppDirect mode, and 192GB of DRAM (6×32GB DIMMs). The CPU has 24 cores (48 hyperthreads) and 35.75MB of L3 cache. The server runs Arch Linux with kernel 5.5.3 and PMDK 1.7. All the code is compiled using GCC 9.2 with all optimization enabled. Threads are pinned to physical cores.

Parameters. For fair comparison, we set CCEH and level hashing to use the same parameters as in their original papers [38, 68]. Our own tests showed these parameters gave the best performance and load factor overall. Level hashing uses 128-byte (two cachelines) buckets. CCEH uses 16KB segments and 64-byte (one cacheline) buckets, with a probing length of four. Dash-EH and Dash-LH use 256-byte (four cachelines) buckets and 16KB segments. Each segment has two stash buckets, making it enough to have four overflow fingerprint slots per bucket so that the overflow counter is rarely positive. Dash-LH uses hybrid expansion with a stride of eight and its first segment array includes 64 segments.

Benchmarks. We stress test each hash table using microbenchmarks. For search operations, we run positive search and negative search: the latter probes specifically non-existent keys. Unless otherwise specified, for all runs we preload the hash table with 10 million records, then execute 190 million inserts (as an insert-only benchmark), 190 million positive search/negative search/delete operations back-to-back on the 200-million-record hash table. For all
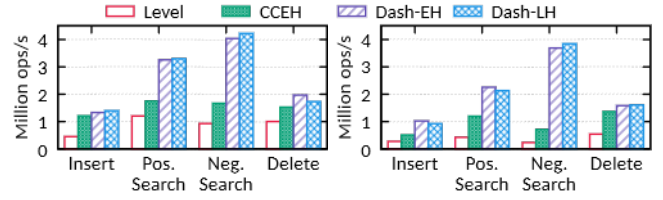
[2]Code downloaded from https://github.com/DICL/CCEH and https://github.com/Pfzuo/Level-Hashing.

[3]We also had to replace `MAP_SHARED_VALIDATE` with `MAP_SHARED` for `MAP_FIXED_NOREPLACE` to work, detailed in our code repository.

Figure 7: Single-thread performance under fixed-length keys (left) and variable-length keys (right).

hash indexes, We use GCC's `std::_Hash_bytes` (based on Murmur hash [1]) as the hash function, which is known to be fast and provides high-quality hashes. Similar to other work [38, 68], we use uniformly distributed random keys in our workloads. We also tested skewed workloads under the Zipfian distribution (with varying skewness) and found all operations achieved better performance benefitting from the higher cache hit ratios on hot keys, and contention is rare because the hash values are largely uniformly distributed. Due to space limitation, we omit the detailed results over skewed workloads. For fixed-length key experiments, both keys and values are 8-byte integers; for variable-length key experiments, we use (pointers to) 16-byte keys and 8-byte values. The variable-length keys are pre-generated by the benchmark before testing.

6.3 Single-thread Performance

We begin with single-thread performance to understand the basic behaviors of each hash table. We first consider a read-only workload with fixed-length keys. Read-only results provide an upper bound performance on the hash tables since no modification is done to the data structure. They directly reflect the underlying design's cache efficiency and concurrency control overhead.

As Figure 7 shows, Dash-EH can outperform CCEH/level hashing by 1.9×/2.6× for positive search. Dash-LH and Dash-EH achieved similar performance because they use the same building blocks, with bounded PM accesses and lightweight concurrency control which reduces PM writes. For negative search, Dash variants achieved more significant improvement, being 2.4×/4.4× faster than CCEH/level hashing. As Section 6.5 shows, this is attributed to fingerprints and the overflow metadata which significantly reduce PM accesses.

For inserts, Dash and CCEH achieve similar performance (~2.5× level hashing). Although CCEH has one fewer cacheline flush per insert than Dash, Dash's bucket load balancing strategy reduces segment splits, improving both performance and load factor. Without the allocation bitmap, CCEH requires a reserved value (e.g., '0') to indicate an empty slot. This design imposes additional restrictions to the application; Dash avoids it using metadata. Level hashing exhibited much lower performance due to more PM reads and frequent lock/unlock operations. It also requires full-table rehashing that incurs many cacheline flushes. For deletes, Dash outperforms CCEH/level hashing by 1.2×/1.9× because of reduced cache misses.

The benefit of Dash is more prominent for variable-length keys. As Figure 7 shows, Dash-EH/LH are 2×/5× faster than CCEH/level hashing for positive search. The differences in negative search are even more dramatic (5×/15×). Again, this results show the effectiveness of fingerprinting. Since all the hash tables store pointers for longer (> 8-byte) keys, the accessing thread has to dereference pointers to load the keys, which is a major source of cache misses. Fingerprinting effectively reduces such indirections. Note that all operations will benefit from this technique, because they either directly query a key (search/delete) or require uniqueness check (insert). For the same reason, Dash-EH outperforms CCEH/level hashing by 2.0×/3.7× for insert, and 1.2×/2.9× for delete.
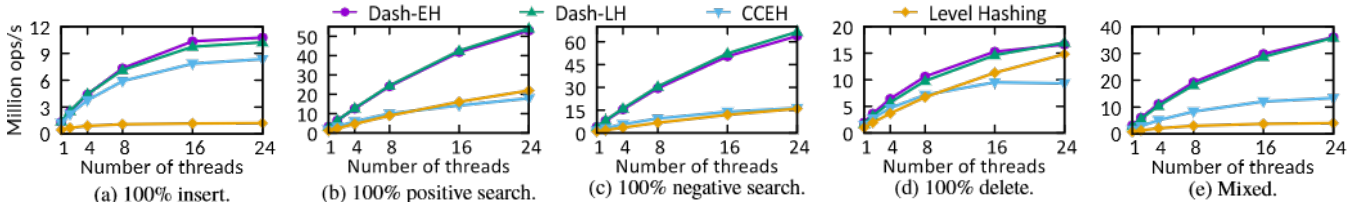
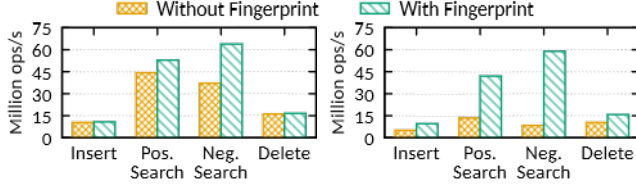**Figure 8:** Throughput under different workloads with a varying number of threads and 8-byte keys and 8-byte values.

Figure 9: Effect of fingerprinting in buckets under fixed-keys (left) and variable-length keys (right).

Figure 10: Effect of overflow metadata (24 threads) with fixed-length keys and two (left) and four (right) stash buckets per segment.

Figure 11: Maximum load factor after adding different techniques.

6.4 Scalability

We test both individual operations and a mixed workload that consists of 20% of insert and 80% of search operations. For the mixed workload, we preload the hash table with 60 million records to allow search operations to access actual data.

Figure 8 plots how each hash table scales under a varying number of threads and fixed-length keys. For inserts, level hashing exhibits the worst scalability mainly due to full-table rehashing, which is time-consuming on PM and blocks concurrent operations. With fingerprinting and bucket load balancing, Dash finishes uniqueness checks quickly and triggers fewer SMOs, with fewer PM accesses and interactions with the PM allocator. Though neither Dash-EH nor Dash-LH scales linearly as inserts inherently exhibit many random PM writes, Dash is the most scalable solution, being up to 1.3×/8.9× faster than CCEH/level hashing for insert operations.

For search operations, Figures 8(b–c) show near-linear scalability for Dash-EH/LH. CCEH falls behind mainly because of its use of pessimistic locking which incurs large amount of PM writes even for read-only workloads (to acquire/release read locks). Level hashing uses a similar design but lock striping [18] makes all the locks likely to fit into the CPU cache. Therefore, although level hashing has lower single-thread performance than CCEH, it still achieves similar performance to CCEH under multiple threads. Delete operations in Dash-EH, Dash-LH, CCEH and level hashing on 24 threads scale and improve over their single-threaded version by 8.4×, 9.8×, 6.1× and 14.7×, respectively. For the mixed workload on 24 threads, Dash outperforms CCEH/level hashing by 2.7×/9.0×.

We observed similar trends (but with widening gaps between Dash-EH/LH and CCEH/level hashing) for workloads using variable-length keys (not shown for limited space). In the following sections, we discuss how each design in Dash impacts its performance.

6.5 Fingerprinting and Overflow Metadata

Fingerprinting is a major reason for Dash to perform and scale well on PM as it greatly reduces PM accesses. We quantify its effect by comparing Dash-EH with and without fingerprinting. Figure 9 shows the result under 24 threads. With fixed-length keys, fingerprinting improves throughput by 1.04/1.19/1.72/1.02× for insert/positive search/negative search/delete. The numbers for variable-length keys are 1.88/3.13/7.04/1.52×, respectively. As introduced in Section 4.3, Dash uses overflow metadata to allow early-stop of

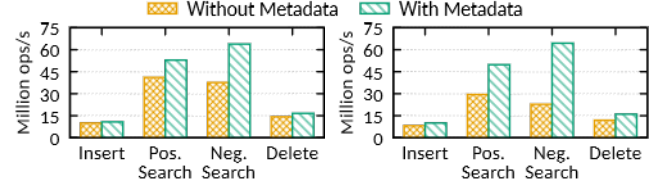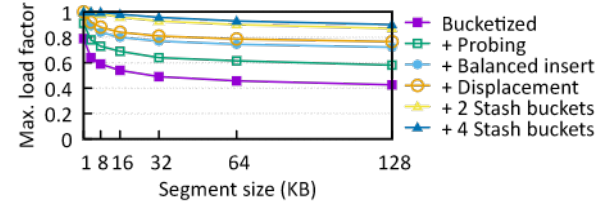search operations to save PM accesses. Figure 10 shows its effectiveness under 24 threads with varying numbers of stash buckets. Dash-EH with two stash buckets outperforms the baseline (no metadata) by 1.07/1.29/1.70/1.16× for insert/positive search/negative search/delete. With more stash buckets added, search performance drops by about 25% without the overflow metadata. With overflow metadata, however, the performance remains stable, as negative search operations can early-stop after checking the overflow metadata without actually probing the stash buckets.

6.6 Load Factor

Now we study how linear probing, balanced insert, displacement and stashing improve load factor. We first measure the maximum load factor of one segment with different sizes. Using larger segments could decrease load factor, but may improve performance by reducing directory size. Real systems must balance this tradeoff.

Figure 11 shows the result. In the figure "Bucketized" represents the simplest segmentation design without the other techniques. The segment can be up to 80% full on 1KB segments but gradually degrades to about 40% full at most as the segment size increases to 128KB. With just one bucket probing added ("+Probing"), the load factor under 128KB segment increases by ∼20%. Balanced insert and displacement can improve load factor by another ∼20%. With stashing, we maintain close to 100% load factor for 1–16KB segments. Dash combines all these techniques to achieve more than twice the load factor than vanilla segmentation with large segments.

To compare different designs more realistically, we observe how load factor changes after a sequence of inserts. We start with an empty hash table (load factor of 0) and measure the load factor after different numbers of records have been added to the hash tables. As shown in Figure 12, the load factor (y-axis) of CCEH fluctuates
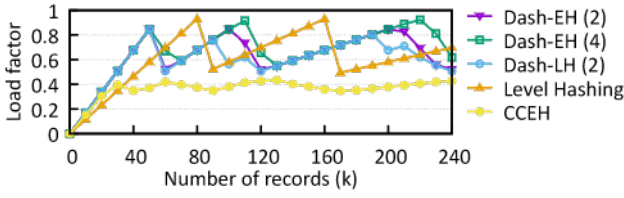
Figure 12: Load factor of different hashing schemes with respect to number of items inserted to the hash table.
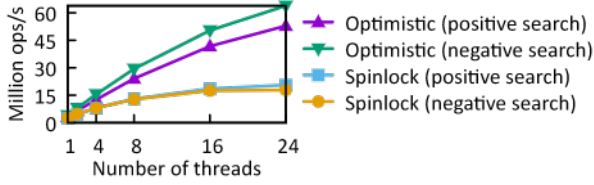

Figure 13: Scalability under different concurrency control strategies (reader-writer spinlocks vs. optimistic locking in Dash-EH).

Table 1: Recovery time (ms) vs. data size. CCEH's recovery time scales with data size. For level hashing and Dash it remains constant because both need a fixed amount of work upon restart.

Hash Table	Number of indexed records (million)					
	40	80	160	320	640	1280
Dash-EH	57	57	57	57	57	57
Dash-LH	57	57	57	57	57	57
CCEH	113	165	262	463	870	1673
Level hashing	53	53	53	53	53	(53)

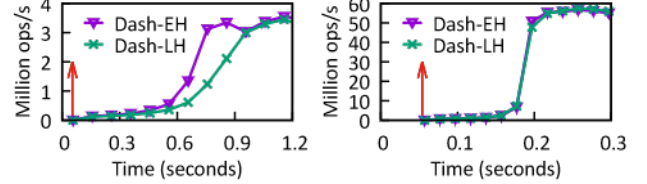**Figure 14:** Throughput under different time points upon restart with one thread (left) and 24 threads (right).

between 35% and 43%, because CCEH only conducts four cacheline probings before triggering a split. As we noted in Section 4.3, long probing lengths increase load factor at the cost of performance, yet short probing lengths lead to pre-mature splits. Compared to CCEH, Dash and level hashing can achieve high load factor because of their effective load factor improvement techniques. The "dipping" indicates segment splits/table rehashing is happening. We also observe that with two stash buckets, denoted as Dash-EH/LH (2), we achieve up to 80% load factor, while the number for using four stash buckets in Dash-EH (4) is 90%, matching that of level hashing.

6.7 Impact of Concurrency Control

As Section 4.4 describes, PM data structures favor lightweight approach such as optimistic locking in Dash, over traditional pessimistic locking. Figure 13 experimentally verifies this point by comparing pessimistic locking (reader-writer spinlocks) and optimistic locking in Dash-EH, under (negative and positive) search workloads. The spinlock based version does not scale well because of extra PM writes needed for manipulating read locks. We repeated the same experiments on DRAM and found that both of them can scale well. This captures yet another important finding that was omitted or not easy to discover in previous emulation based studies.

6.8 Recovery

It is desirable for persistent hash tables to recover instantly after a crash or clean shutdown to reduce service downtime. We test recovery time by first loading a certain number of records and then killing the process and measuring the time needed for the system to be able to handle incoming requests. Table 1 shows the time needed for each hash table to get ready for handling incoming requests under different data sizes. The recovery time for Dash-EH/LH and level hashing are at sub-second level and does not scale as data size increases, effectively achieving instant recovery. For Dash-EH/LH the only needed work is to open the PM pool that is backing the hash table, and then read and possibly set the values of two variables. For 1280M data, level hashing requires an allocation size greater than the maximum allowed by PMDK allocator (15.998GB). However, it also only needs a fixed amount of work to open the PM pool during the recovery, so we expect its recovery time would remain the same (53ms) under larger data sizes. The recovery time for CCEH is

linearly proportional to the size of the indexed data because it needs to scan the entire directory upon recovery. As data size increases, so is the directory size, requiring more time on recovery.

Dash's lazy recovery may impact runtime performance. We measure this effect by recording the throughput over time of Dash-EH/LH once it is instantly recovered. The hash table is pre-loaded with 40 million records. We then kill the process while running a pure insert workload and restart the hash table and start to issue positive search operations to observe how throughput changes over time. The result is shown in Figure 14; the red arrow indicates the time point when Dash is back online to be able to serve new requests. The throughput is relatively low at the beginning: 0.1–0.3 Mops/s under one thread in Figure 14(left), and 0.6 Mops/s under 24 threads in Figure 14(right). Using more threads can help throughput to return to normal earlier, as multiple threads could hit different segments and work on the rebuilding of metadata or concluding SMOs in parallel. Throughput returns to normal in 0.2 seconds under 24 threads, while the number under one thread is 0.9 s.

6.9 Impact of PM Software Infrastructure

It has been shown that PM programming infrastructure can be a major overhead due to reasons such as page faults and cacheline flushes [22, 31]. We quantify its impact by running the same insert benchmark in Section 6.4 under two allocators (PMDK vs. a customized allocator) and two Linux kernel versions (5.2.11 vs. 5.5.3). Our customized allocator pre-allocates and pre-faults PM to remove page faults at runtime. Though not practical, it allows us to quantify the impact of PM allocator; it is not used in other experiments. As Figure 15(left) shows, Dash-EH is not very sensitive to allocator performance under different kernel versions as its allocation size (single segment, 16KB) is fixed and not huge. Dash-LH in Figure 15(right), however, exhibited very low performance using PMDK allocator on kernel 5.2.11 ($\sim$ 25% the number under 5.5.3). We found the reason was a bug[4] in kernel 5.2.11 that can cause large PM allocations to fall back to use 4KB pages, instead of 2MB huge pages (PMDK default). This led to many more page faults and OS scheduler activities, which impact Dash-LH the most, as linear hashing inherently

---
[4]Caused by a patch discussed at lkml.org/lkml/2019/7/3/95, fixed by patch at lkml.org/lkml/2019/10/19/135 (5.3.8 and newer). More details are available in our code repository.
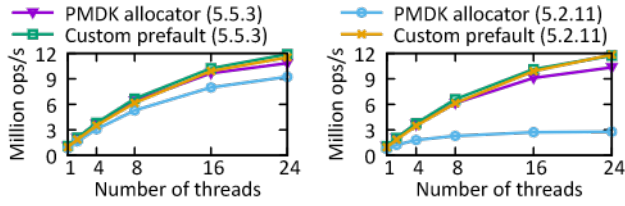
Figure 15: Impact of PM allocator and OS support on Dash-EH (left) and Dash-LH (right).

requires multiple threads compete for PM allocation during split operations and slow PM allocation could block concurrent requests. The increased number of page faults also impacted recovery performance. For instance, under 160 million records, it took CCEH ~10× longer on kernel 5.2.11 than the number in Table 1 to recover.

These results highlight the complexity of PM programming and call for careful design and testing involving both userspace (e.g., allocator) and OS support. We believe it is necessary as the PM programming stack is evolving rapidly while practitioners and researchers have started to rely on them to build PM data structures.

7. RELATED WORK

Dash builds upon many techniques from prior in-memory and PM-based hash tables, tree structures and PM programming tools.

In-Memory Hash Indexes. Section 2.2 has covered extendible hashing [12] and linear hashing [29, 36], so we do not repeat here. Cuckoo hashing [44] achieves high memory efficiency through displacement: a record can be inserted into one of the two buckets computed using two independent hash functions; if both buckets are full, a randomly-chosen record is evicted to its alternative bucket to make space for the new record. The evicted record is inserted in the same way. MemC3 [13] proposes a single-writer, multi-reader optimistic concurrent cuckoo hashing scheme that uses version counters with a global lock. MemC3 [13] also uses a tagging technique which is similar to fingerprinting to reduce the overhead of accessing pointers to variable-length keys. FASTER [5] further optimizes it by storing the tag in the unused high order 16 bits in each pointer. `libcuckoo` [33] extends MemC3 to support multi-writer. Cuckoo hashing approaches may incur many memory writes due to consecutive cuckoo displacements. Dash limits the number of probings and uses optimistic locking to reduce PM writes.

Static Hashing on PM. Most work aims at reducing PM writes, improving load factor and reducing the cost of full-table rehashing. Some proposals use multi-level designs that consist of a main table and additional levels of stashes to store records that cannot be inserted into the main table. PFHT [10] is a two-level scheme that allows only one displacement to reduce writes. It uses linked lists to resolve collisions in the stash, which may incur many cache misses during probing. Path hashing [67] further organizes the stash as an inverted complete binary tree to lower search cost. Level hashing [68] is a two-level scheme that bounds the search cost to at most four buckets. Upon resizing, the bottom-level is rehashed to be 4× size of the top-level table, and the previous top level becomes the new bottom level. Compared to cuckoo hashing, the number of buckets needed to probe during a lookup is doubled. Dash also uses stashes to improve load factor, but most search operations only need to access two buckets thanks to the overflow metadata.

Dynamic Hashing on PM. CCEH [38] is a crash-consistent extendible hashing scheme which avoids full-table rehashing [12]. To improve search efficiency, it bounds its probing length to four cachelines, but this can lead to low load factor and frequent segment splits.

CCEH's recovery process requires scanning the directory upon restart, thus sacrifices instant recovery. Prior proposals often use pessimistic locking [38, 68] which can easily become a bottleneck due to excessive PM writes when manipulating locks. The result is even conflict-free search operations cannot scale. NVC-hashmap [51] is a lock-free, persistent hash table based on split-ordered lists [52]. Although the lock-free design can reduce PM writes, it is hard to implement; the linked list design may also incur many cache misses. Dash solves these problems with optimistic locking that reduces PM writes and allows near-linear scalability for search operations.

Range Indexes. Most range indexes for PM are B+-tree or trie variants and aim to reduce PM writes [2, 6, 7, 19, 30, 43, 59, 60, 64]. An effective technique is unsorted leaf nodes [6, 7, 43, 64] at the cost of linear scans, while hash indexes mainly reduce PM writes by avoiding consecutive displacements. FP-tree [43] proposes fingerprints in leaf nodes to reduce PM accesses; Dash adopted it to reduce unnecessary bucket probing and efficiently support variable-length keys. Some work [43, 60] places part of the index in DRAM (e.g., inner nodes) to improve performance. This trades off instant recovery as the DRAM part has to be rebuilt upon restart [31]. The same tradeoff can be seen in hash tables by placing the directory in DRAM. With bucket load balancing techniques, Dash can use larger segments and place the directory in PM, avoiding this tradeoff.

PM Programming. PM data structures rely heavily on userspace libraries and OS support to easily handle such issues as PM allocation and space management. PMDK [20] is so far the most popular and comprehensive library. An important issue in these libraries is to avoid leaking PM permanently. A common solution [4, 20, 42, 50] is to use an allocate-activate approach so that the allocated PM is either owned by the application or the allocator upon a crash. At the OS level, PM file systems provide direct access (DAX) to bypass caches and allow pointer-based accesses [35]. Some traditional file systems (e.g., ext4 and XFS) have been adapted to support DAX. PM-specific file systems are also being proposed to further reduce overhead [8, 23, 28, 47, 57, 61]. We find support for PM programming is still in its early stage and evolving quickly with possible bugs and inefficiencies as Section 6.9 shows. This requires careful integration and testing when designing future PM data structures.

8. CONCLUSION

Persistent memory brings new challenges to persistent hash tables in both performance (scalability) and functionality. We identify that the key is to reduce both unnecessary PM reads and writes, whereas prior work solely focused on reducing PM writes and ignored many practical issues such as PM management and concurrency control, and traded off instant recovery capability. Our solution is Dash, a holistic approach to scalable PM hashing. Dash combines both new and existing techniques, including (1) fingerprinting to reduce PM accesses, (2) optimistic locking, and (3) a novel bucket load balancing technique. Using Dash, we adapted extendible hashing and linear hashing to work on PM. On real Intel Optane DCPMM, Dash scales with up to ~3.9× better performance than prior state-of-the-art, while maintaining desirable properties, including high load factor and sub-second level instant recovery.

9. REFERENCES

[1] A. Appleby. Murmurhash.
https://sites.google.com/site/murmurhash/.

[2] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5):553–565, 2018.

[3] BerkeleyDB. Access Method Configuration.
https://docs.oracle.com/cd/E17076_02/html/programmer_reference/am_conf.html#id3931891.

[4] K. Bhandari, D. R. Chakrabarti, and H. Boehm. Makalu: fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 677–694, 2016.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 275–290, 2018.

[6] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 21–31, 2011.

[7] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 133–146, 2009.

[9] R. Crooke and M. Durcan. A revolutionary breakthrough in memory technology. *3D XPoint Launch Keynote*, 2015.

[10] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW 2015, Monterey, California, USA, October 4, 2015*, pages 1:1–1:9, 2015.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):18, June 1984.

[12] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing–a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, Sept. 1979.

[13] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[14] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2004.

[15] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, Dec. 1992.

[16] S. Ghemawat and J. Dean. LevelDB. 2019.
https://github.com/google/leveldb.

[17] W. G. Griswold and G. M. Townsend. The design and implementation of dynamic hashing for sets and tables in Icon. *Softw., Pract. Exper.*, 23(4):351–367, 1993.

[18] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[19] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[20] Intel. Persistent Memory Development Kit. 2018.
http://pmem.io/pmdk/libpmem/.

[21] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. 2015.

[22] 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, 2019.

[23] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 494–508, 2019.

[24] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi. SLM-DB: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019.*, pages 191–205, 2019.

[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. *SIGMOD*, pages 691–706, 2015.

[26] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, 2013.

[27] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[28] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. E. Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 460–477, 2017.

[29] P.-A. Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, Apr. 1988.

[30] 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 *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, 2017. USENIX Association.

[31] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *PVLDB*, 13(4):574–587, 2019.

[32] J. J. Levandoski, D. B. Lomet, S. Sengupta, A. Birka, and C. Diaconu. Indexing on modern hardware: hekaton and beyond. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 717–720, 2014.

[33] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 27:1–27:14, 2014.

[34] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In

Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, pages 21–35, 2017.

[35] Linux. Direct access for files. 2019. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[36] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 212–223. VLDB Endowment, 1980.

[37] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, pages 133–148, 2016.

[38] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Feb. 2019.

[39] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 37:1–37:16, 2017.

[40] Oracle. Architectural overview of the Oracle ZFS storage appliance. 2015. https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overview-zfsa-2099942.pdf.

[41] Oracle. MySQL. 2019. https://www.mysql.com/.

[42] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 10(11):1166–1177, 2017.

[43] 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 the 2016 International Conference on Management of Data, SIGMOD*, pages 371–386, 2016.

[44] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[45] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[46] PostgreSQL Global Development Group. PostgreSQL. 2019. http://www.postgresql.org/.

[47] D. S. Rao, S. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 15:1–15:15, 2014.

[48] Redis Labs. Redis. 2019. https://redis.io.

[49] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 231–244, 2002.

[50] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*, pages 61–72, 2015.

[51] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, IMDM '15, pages 4:1–4:8, 2015.

[52] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[53] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[54] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, pages 18–32, 2013.

[55] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555, 2018.

[56] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019.*, pages 12:1–12:7, 2019.

[57] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 14:1–14:14, 2014.

[58] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[59] T. Wang, J. Levandoski, and P.-A. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, April 2018.

[60] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 349–362, 2017.

[61] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 323–338, 2016.

[62] S. Xu, S. Lee, S. W. Jun, M. Liu, J. Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. *PVLDB*, 10(4):301–312, 2016.

[63] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, February 24-27*, 2020.

[64] 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 *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[65] D. Zhang and P.-A. Larson. Lhlf: Lock-free linear hashing (poster paper). In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 307–308, New York, NY, USA, 2012. ACM.

[66] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016*

International Conference on Management of Data, SIGMOD
'16, pages 1567–1581, New York, NY, USA, 2016. ACM.

[67] P. Zuo and Y. Hua. A write-friendly and cache-optimized
hashing scheme for non-volatile memory systems. *IEEE
Transactions on Parallel Distributed Systems (TPDS)*,
29(5):985–998, 2018.

[68] P. Zuo, Y. Hua, and J. Wu. Write-optimized and
high-performance hashing index scheme for persistent
memory. In *13th USENIX Symposium on Operating Systems
Design and Implementation (OSDI 18)*, pages 461–476, Oct.
2018.