Scaling Dynamic Hash Tables on Real 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 piecewise and partial solutions that side-stepped 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 server with Optane DCPMM, compared to state-of-the-art, Dash 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, 14] and key-value stores [3, 15]. Persistent memory (PM) such as 3D XPoint [1] 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 [2, 7] would not reap its full benefits, necessitating a departure from conventional designs.

 This is a minor revision of the paper entitled "Dash: Scalable Hashing on Persistent Memory", published in PVLDB, Vol. 13, No. 8, ISSN 2150-8097. DOI: https://doi.org/10.14778/3389133.3389134
*Work partially performed while at Simon Fraser University.

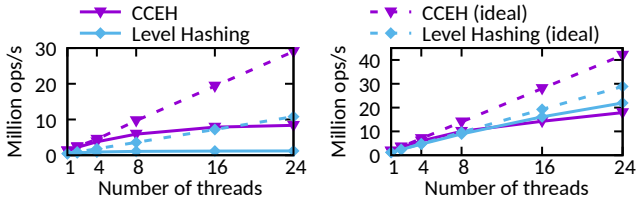

Figure 1: Throughput of state-of-the-art PM hash tables (CCEH [10] and Level Hashing [22]) for insert (left) and search (right) operations on Optane DCPMM. Neither matches the expected scalability.

1.1 Hashing on PM: Not What You Assumed!

There have been a new breed of hash tables specifically designed for PM [10, 17, 22] 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 [10, 22] 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, or even read-only search operations. Corroborating with recent work [8, 20], we find the main culprit is Optane DCPMM's limited bandwidth, which is ∼3–14× lower than DRAM's. Although the server is fully populated to provide the maximum possible bandwidth, excessive PM accesses can still easily saturate and prevent the system from scaling. We describe two main 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. Reducing PM writes has been a main theme in recent work, but many existing solutions incur more PM reads. We note that it is also imperative to reduce 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 [20]. 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, multiple buckets may

have to be searched, incurring many cache misses and PM reads when comparing keys.

Heavyweight Concurrency Control. Most prior work side-stepped the impact of concurrency control. Bucket-level locking has been widely used [10, 22], but it incurs additional PM writes to acquire/release read locks, further pushing bandwidth consumption towards the limit. Lock-free designs [17] can avoid PM writes for read-only probing operations, but are notoriously hard to get right, more so on PM when safe persistence is necessary [19].

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 was often traded off for performance (though scalability is still an issue on real PM). (1) Indexes could occupy more than 50% of memory capacity [21], 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) [10]. 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 them. (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-stepped 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 [12] 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 uses 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 design Dash to only perform a constant amount of work upon restart (reading and possibly writing a one-byte counter) and amortize the "real" recovery work to runtime. Compared to prior work that handles PM programming issues in ad hoc ways, Dash uses well-defined PM programming models (PMDK [4], 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. In this paper, we focus on dynamic hashing and apply Dash to two classic and widely-used approaches: extendible hashing [2, 10] and linear hashing [7]. Evaluation using a 24-core Intel Xeon Scalable CPU and 768GB of Optane DCPMM shows that compared to existing state-of-the-art [10, 22], Dash achieves up to ∼3.9× better performance on realistic workloads, up to over 90% of load factor with high space utilization and the ability to instantly recover in 57ms regardless of data size. Our implementation is open-source at: https://github.com/baotonglu/dash.

2. BACKGROUND

We first give necessary background on PM hardware 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 at a price lower than DRAM's. Similar to other work [10,22], we leverage its AppDirect mode, as it provides more flexibility and persistence guarantees.

System Architecture. Current mainstream CPU architectures (e.g., Intel Cascade Lake) place DRAM and PM behind multiple levels of *volatile* CPU caches. Data is not guaranteed to be persisted in PM until a cacheline flush instruction (e.g., `CLWB` [5]) is executed or other events that implicitly cause cacheline flush occur. Writes to PM may be reordered, requiring fences to avoid undesirable reordering. The application (hash tables in our case) must issue fences and cacheline flushes to ensure correctness. 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 (not necessarily the DCPMM media), it is considered persistent.

Future CPU and DCPMM generations (e.g., the upcoming Intel Ice Lake CPUs) may feature extended ADR (eADR) which further includes the CPU cache in the ADR domain [16], effectively providing persistent CPU caches and thus eliminating the need for cacheline flushes (fences are still needed). The current implementation of Dash still issues cacheline flushes, however, our preliminary experiments that skip cacheline flushes on existing Cascade Lake CPUs showed eADR's potential impact is very small for hash tables. We attribute the reason to hash table's inherently random access patterns.

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 [20], 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 (also ensure persistence) once the data reaches the ADR domain at the memory controller rather than when reaching the PM 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 in data structures with inherent randomness, e.g., hash tables). Tests also showed that the bandwidth of DCPMM depends

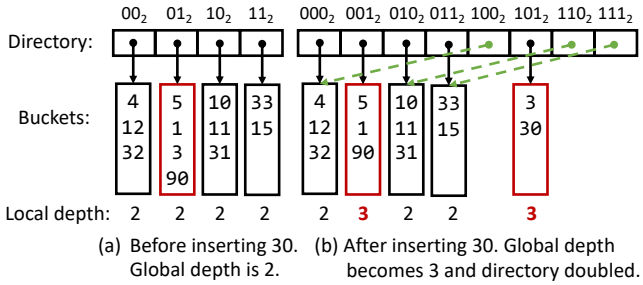

Figure 2: An example of extendible hashing.

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, DCPMM exhibits very limited performance for small, random accesses [20], which are inherent access pattern for hash tables. These properties exhibit a stark contrast to prior estimates [13, 18], 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.

2.2 Dynamic Hashing

Now we give an overview of extendible hashing [2] and linear hashing [7]. We focus on their memory-friendly versions which PM-adapted hash tables were based upon.

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. 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*. The number of directory entries pointing to one bucket is $2^{global\ depth-local\ depth}$. 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 01_2 in Figure 2(a)), then the directory needs to be doubled to accommodate the new bucket. Figure 2 shows an example of splitting the full bucket 01_2 when an inserting key 30 is hashed into that bucket. After bucket splitting, *local depth* of the splitting bucket needs to be properly updated. Choosing a proper hash function that evenly distributes keys to all buckets is an important but orthogonal problem.

Linear Hashing. In-memory linear hashing takes a similar approach to organizing buckets using a directory with entries pointing to individual buckets [7]. 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. Compared with extendible hashing, linear hashing could have smaller directory size by proper organization [7].

2.3 Dynamic Hashing on PM

To reduce PM accesses on dynamic extendible hashing, CCEH [10] groups buckets into larger *segments*. Each directory entry then points to a segment which consists of a fixed number of buckets. This design reduces the size of the directory, 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 premature 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 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 involves lots of PM writes. Existing approaches such as CCEH side-step PM management issues surrounding segment split, having the risk of permanent PM leaks.

3. DESIGN PRINCIPLES

The discussions in Section 2 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 avoid unnecessary PM reads to conserve bandwidth and alleviate the impact of high end-to-end read latency.
- **Lightweight Concurrency.** Dash must scale well on multicore machines with persistence guarantees. Given the limited bandwidth, concurrency control must be lightweight to incur not 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 instant 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 [7, 10], 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

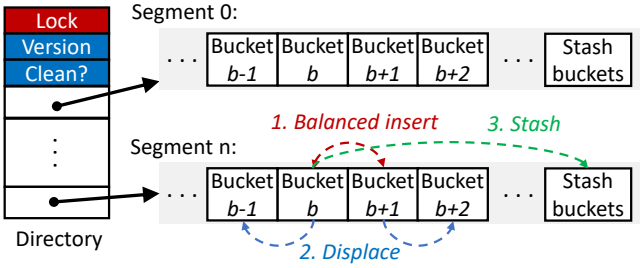

Figure 3: Overall architecture of Dash-EH.

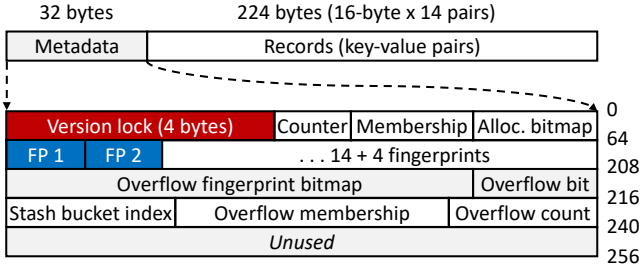

Figure 4: Dash-EH bucket layout. The metadata optimizes probing and load factor, followed by records.

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 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, which allows us to store 14 records 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. What follows are structures such as fingerprints and overflow metadata to accelerate probing and improve load factor.

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 [12] 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.

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. Figure 3 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). The rationale behind is to improve load factor by amortizing the load of hot buckets while limiting PM accesses (at most two buckets).

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. 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 bucket $b+2$. 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 4) to indicate which records could be selected for displacement, accelerating this process.

Stashing. As shown 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 b nor the probing bucket $b+1$, 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 and overflow metadata (bits 208-240) are reserved for overflow records stored

in stash buckets. These metadata could indicate whether the searching key exists in the stash area, allowing early avoidance of access to stash buckets (and save PM bandwidth). We omit details here for space limitation. 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 [6]. 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). The inserting thread will acquire bucket-level locks for the target and probing buckets by trying the `compare-and-swap` (`CAS`) instruction [5] to set the lock bit. 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.

4.5 Support for Variable-Length Keys

Dash stores pointers to variable-length keys, which is a common approach [10, 12, 22]. 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 [12].

4.6 Record Operations

Now we present 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. The inserting thread first writes and persists the new record in bucket, and then set up the metadata (fingerprint, allocation bitmap, counter and membership). The `CLWB` and fence are then issued 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. 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 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).

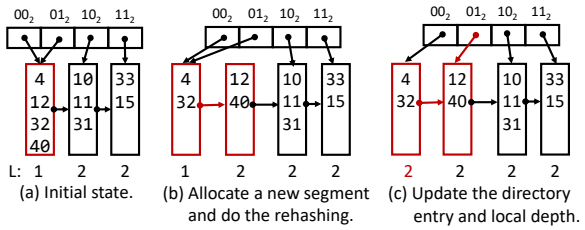

Figure 5: Segment split in Dash-EH; the global depth is 2.

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. If neither bucket contains the record, it might be stored in a stash bucket. It will first probe the overflow metadata area of bucket b and access the stash buckets if necessary.

Delete. To delete a record in the 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, we also need to clear the corresponding overflow metadata stored in the target bucket.

4.7 Structural Modification Operations

After a thread exhausted all the options to insert a record into a bucket, it triggers a segment split that may expand the directory. To split a segment S, we (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 steps cause the structure of the hash table to change and must be made crash consistent on PM while maintaining high performance.

For crash consistency, Dash-EH chains all segments using side links and each segment has a `state` variable that indicates 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 Dash-EH uses the most significant bits (MSBs) of hash values to address and organize segments and buckets, which reduces cacheline flushes in the directory during splits [10]. 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 [4]) 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; this allows Dash-EH to employ 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` (boolean) marker shown in Figure 3, and a per-segment version number. The `clean` marker denotes whether the system was shutdown cleanly; V tells whether recovery (during runtime) is needed. Upon restart, if `clean` is false (no clean shut down), 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. Recovering a segment needs four steps: (1) clear bucket locks, (2) remove duplicate records caused by displacement, (3) rebuild overflow metadata, and (4) continue the ongoing SMO. With such lazy recovery approach, a segment is not recovered until it is accessed.

5. Dash FOR LINEAR HASHING

We present Dash-LH, Dash-enabled linear hashing that uses the building blocks discussed previously. We focus on the high-level design decisions specific to linear hashing; more details can be found in our original VLDB paper [9].

Figure 6 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 full, full segments need to be able to temporarily accommodate overflow records, e.g., using linked lists. Traversing linked lists would incur many cache misses, which is a huge penalty for PM hash tables. We leverage stashing in Dash and use an adjustable number of stash buckets. In addition to a fixed number of stash buckets per segment, we store a linked list of stash buckets. A segment split is triggered whenever a stash bucket is allocated to accommodate overflow records. 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 alleviate the performance penalty brought by the need to search stash buckets. To reduce directory size for better cache locality, we also introduce a hybrid expansion scheme which increases segment size exponentially. Overall, as we show in Section 6, Dash-LH can also achieve near-linear scalability on realistic workloads.

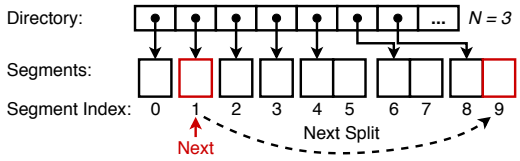

Figure 6: Overall design of Dash-enabled linear hashing.

6. EVALUATION

We evaluate Dash and compare it with state-of-the-art PM hash tables. 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 recovers instantly with a small, constant amount of work upon restart, reducing service downtime.

Implementation. We implemented Dash-EH/LH using PMDK [4], which provides primitives for crash-safe PM management. The other hash tables under comparison (CCEH [10] and level hashing [22]) were both proposed based on DRAM emulation. We ported them to run on Optane DCPMM using their original code and PMDK; details are available in our original VLDB paper [9].

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 [10, 22]. 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.

Benchmarks. We stress test each hash table using microbenchmarks. 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. Similar to other work [10, 22], we use uniformly distributed random keys in our workloads. Due to space limitation, we omit the results over skewed workloads but all operations achieved similar and even better performance for higher cache hit ratios. For fixed-length key experiments, both keys and values are 8-byte

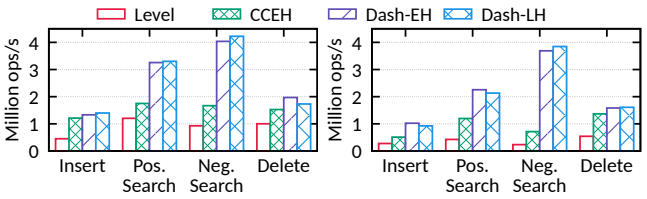

Figure 7: Single-thread performance under fixed-length keys (left) and variable-length keys (right).

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.

6.1 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 achieved more significant improvement, benefitting from fingerprints and the overflow metadata.

For inserts, Dash and CCEH achieved 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. Level hashing exhibited much lower performance due to more PM reads. 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 numbers are even greater (5×/15×) for negative search. These results again show the effectiveness of fingerprinting which all operations will benefit from, because they either directly query a key (search/delete) or require uniqueness check (insert).

6.2 Scalability

We test both individual operations and a mixed workload (20% insert and 80% 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 insert operations, 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. Although 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 search operations, Figures 8(b–c) show near-linear scalability for Dash-EH/LH. CCEH falls behind mainly due to 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 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).

6.3 Load Factor

To compare different designs 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 9, the load factor (y-axis) of CCEH fluctuates 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.

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

6.5 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 [8, 20]. We quantify its impact by running the same insert benchmark in Section 6.2 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. An interesting finding is that Dash-LH

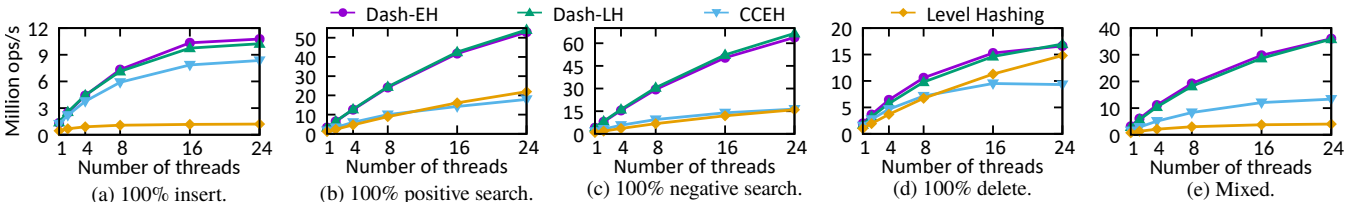

Figure 8: Throughput under different workloads with a varying number of threads and 8-byte keys and 8-byte values.

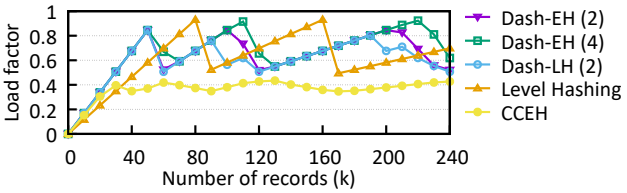

Figure 9: Load factor of different hashing schemes with respect to number of records inserted to the hash table.

Table 1: Recovery time (ms) vs. data size. CCEH's recovery time scales with data size whereas Dash and level hashing's remain constant.

Hash Table	Number of 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)

exhibited very low performance using PMDK allocator on Linux kernel 5.2.11 ($\sim$ 25% the number under 5.5.3). The reason was a bug in Linux kernel 5.2.11 that can cause large PM allocations to fall back to use 4KB pages, instead of 2MB huge pages (PMDK default).

Such results highlight the complexity of PM programming and call for careful design and testing in user and kernel spaces, given that the PM programming stack is evolving rapidly while practitioners and researchers have started to rely on them to build PM data structures.

7. 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 $\sim$3.9$\times$ better performance than prior state-of-the-art, while maintaining desirable properties, including high load factor and sub-second level instant recovery.

8. REFERENCES

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

[2] R. Fagin et al. Extendible hashing–a fast access method for dynamic files. *ACM TODS*, pages 315–344, 1979.

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

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

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

[6] H. T. Kung et al. On optimistic methods for concurrency control. *ACM TODS*, pages 213–226, 1981.

[7] P.-A. Larson. Dynamic hash tables. *CACM*, 1988.

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

[9] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *PVLDB*, 13(8):1147–1161, 2020.

[10] M. Nam et al. Write-optimized dynamic hashing for persistent memory. *FAST*, pages 31–44, Feb. 2019.

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

[12] I. Oukid et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. *SIGMOD*, 2016.

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

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

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

[16] S. Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, 2020.

[17] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. *IMDM*, pages 4:1–4:8, 2015.

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

[19] T. Wang, J. Levandoski, and P.-A. Larson. Easy lock-free indexing in non-volatile memory. *ICDE*, 2018.

[20] J. Yang et al. An empirical guide to the behavior and use of scalable persistent memory. *FAST*, 2020.

[21] H. Zhang et al. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. *SIGMOD*, pages 1567–1581, 2016.

[22] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. *OSDI*, pages 461–476, Oct. 2018.