PMap: A Non-volatile Lock-free Hash Map with Open Addressing

1st Kenneth Lamar
Department of Computer Science
University of Central Florida
Orlando, United States
kenneth@knights.ucf.edu

2nd Christina Peterson
Department of Computer Science
University of Central Florida
Orlando, United States
clp8199@knights.ucf.edu

3rd Damian Dechev
Department of Computer Science
University of Central Florida
Orlando, United States
dechev@cs.ucf.edu

4th Roger Pearce
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, United States
pearce7@llnl.gov

5th Keita Iwabuchi
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, United States
iwabuchi1@llnl.gov

6th Peter Pirkelbauer
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, United States
pirkelbauer2@llnl.gov

***Abstract*—Non-volatile memory (NVM) is an emerging memory technology that provides data persistence and higher densities than conventional DRAM. The release of Intel Optane DC memory makes NVM a practical and testable technology. Hash maps are fundamental data structures that associatively map keys to values, offering constant time lookup. In this work, we designed a scalable, persistent hash map, PMap, optimized around large graph processing workloads. PMap is a lock-free non-volatile hash map with open addressing. Open addressing offers low memory overhead and improved cache locality when compared with node-based alternatives. Lock-freedom ensures scalable performance, and its nonblocking nature enables log-free persistence. Our hash map is supported by a non-blocking, parallel resize, which allows operations to be performed by other threads during a resize. In our performance tests, we found that our design outperformed state-of-the-art alternatives, averaging 3122x faster under Optane DC.**

***Index Terms*—hash map, hash table, key-value store, non-volatile memory, lock-free**

I. Introduction

Hash maps are often a core structure for graph representation in memory. Our work is motivated by the need for data structures suitable for large-scale graph analytics operations [1]. Graph analytics are used for many tasks, including page ranking, pattern matching, and clustering. These workloads may involve billions of vertices or more, making them difficult to manage in-memory [2]. DRAM capacities are too small to store large graphs entirely in memory, so flash memory is conventionally used as a supplement. Flash memory is block-based, requiring custom algorithms to efficiently support large-scale data processing. Further, DRAM is volatile, meaning unexpected system failure can result in data loss during time-consuming graph construction work or graph updates. NVM offers byte-addressable storage with large capacities and data persistence, making it a good fit for this target use-case.

Existing NVM hash maps are traditionally lock-based. These designs have limited scalability due to heavy lock contention at scale. Non-blocking designs use atomic operations to synchronize threads without locks, improving scalability. Lock-freedom is a non-blocking progress guarantee that ensures that at least one thread always makes progress.

Lock-free algorithms are already difficult to reason about, so it follows that adding NVM support to lock-free algorithms is even more challenging. While atomic primitives maintain thread operation order, they do not enforce memory order; writes on a thread may only affect the volatile CPU cache and later persist to NVM without regard to other threads, resulting in an inconsistent recovery state. Enforcing memory ordering is done by inserting fences between instructions to prevent instruction reordering and by adding flush instructions to explicitly evict cache lines to memory in the correct order. Enforcing memory order while minimizing the number, and thus impact, of flush and fence operations remains an open problem.

While lock-free persistent hash maps have been developed, most perform poorly due to a heavy use of pointer dereferences. Open addressing enables the contiguous placement of keys and values to improve cache locality and eliminate the memory overheads of pointer storage. Developing a persistent lock-free design with open addressing is rare, with only one existing design identified, concurrent level hashing (clevel) [3]. While clevel uses open addressing, keys and values are still stored via pointer, reducing the cache locality typically associated with open addressing.

In this work, we propose PMap, a **P**ersistent concurrent hash **Map** with open addressing and non-blocking parallel resizing. Our core design is based on Click's hash map [4]. Click's hash map is lock-free and offers non-blocking parallel resizing. It uses open addressing via linear probing, resulting in excellent cache locality. To provide persistence, our design uses a modified version of the link-and-persist approach [5], [6]. While this approach is traditionally used to persist pointers, our modification enables link-and-persist to be used in-place, persisting keys and values directly. This is crucial, as descriptors, logs, or separate object allocations may require pointer dereferences, which undermines the cache locality benefits of pure open addressing. As our design uses open addressing, table levels can be contiguously allocated for resize. This simplifies persistence and recovery, as whole tables can be mapped and verified to complete recovery with no need to map smaller persistent objects. All these design choices result in a hash map we believe to be well-suited to graph processing while also offering excellent performance and functionality for general use cases. In summary, PMap distinguishes itself from prior work with the following contributions.

- A new persistent memory design and implementation: We integrate persistence into the Click hash table to provide a practical, scalable design.
- An efficient persistence design that can persist keys and value in-place: We repurpose link-and-persist for use on keys and values directly, rather than via copy-on-write (CoW). This approach also eliminates the need for descriptors, logs, or additional allocated objects.
- A new approach to resizing in persistent memory: Unlike non-blocking NVM alternatives, our resize supports size reductions as well as expansions.
- Performance testing and results: We run practical tests against state-of-the-art alternatives and find our design to be an average 3122x faster than the previous state-of-the-art, clevel in our alternating test.
- Improvements over the state-of-the art: When compared to the best and only known lock-free persistent hash table with open addressing [3], our approach offers reduced pointer space, reduced dereference overhead, and improved cache locality by inserting keys and values in-place, support for capacity reductions by design, reduced memory fragmentation, conditional value replacement, and no CoW operations, resulting in a significant performance advantage.

II. Design

A. Design Requirements

Our hash map was designed to perform well in large-scale graph analytics. Based on this primary use case, we established the following design requirements.

- Optimized for read-heavy workloads: Analysis of graphs is typically read-heavy. We use a persistence approach that requires no flush or fence operations when reading and offers fast table searches.
- Prioritize runtime performance over recovery performance: Recovery is rare, so persisting less information at the cost of a longer recovery is acceptable.
- Compact representation with few cache misses: Open addressing enables the contiguous placement of keys and values to improve cache locality and eliminate the memory overheads of pointer storage.
- Low memory management overhead: When possible, we allocate large table chunks instead of small, per key-value objects, to reduce the number of allocations.

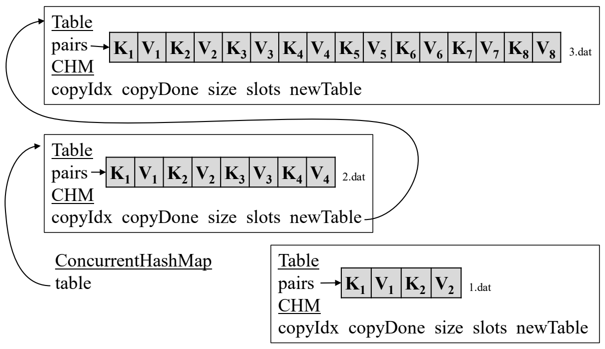


Table
pairs
CHM
copyIdx copyDone size slots newTable
K_1 V_1 K_2 V_2 K_3 V_3 K_4 V_4 K_5 V_5 K_6 V_6 K_7 V_7 K_8 V_8
3.dat
Table
pairs
CHM
copyIdx copyDone size slots newTable
K_1 V_1 K_2 V_2 K_3 V_3 K_4 V_4
2.dat
ConcurrentHashMap
table
Table
pairs
CHM
copyIdx copyDone size slots newTable
K_1 V_1 K_2 V_2
1.dat

Fig. 1: The data layout of the hash table. Underlined text indicates class names, grey boxes contain data stored in persistent memory, and arrows indicate pointers. CHM contains auxiliary counters (`copyIdx`, `copyDone`, `size`, `slots`) and a pointer to the new table (`newTable`). `copyIdx` tracks the number work chunks claimed by resizers while `copyDone` records the number of completed work chunks. `size` tracks the number of active key-value pairs and `slots` is the number of unclaimed locations to place elements.

The current state-of-the-art, clevel [3] generally meets these goals well. Clevel uses a two hash solution to improve performance at high load factors, reducing the need to resize and thus allowing for a compact representation. It uses link-and-persist [5], [6] to minimize the number of flush and fence operations and provide recovery, open addressing to reduce the need for pointers, and dynamic persistent memory allocation to manage object pairs. However, the paper does not detail how recovery works, the design still uses pointers when accessing key-value objects (as they would otherwise be limited to the size of an atomic), and the dynamic allocation of object pairs increases the complexity and overhead of persistence. With the limits of clevel in mind, we worked to develop an improved hash table design.

B. Data Layout

As illustrated in Fig. 1, each ConcurrentHashMap (PMap) consists of a `table` pointer, which points to the lowest level table that contains valid elements. Within each table is a `pairs` pointer, which points to a chunk of contiguously allocated persistent memory to store keys (K) and values (V). Each key and value in the illustration has a number to indicate which key and value are associated with each other in the structure. Each table level is named in the direct access (DAX) filesystem in the order it was allocated (1.dat, 2.dat, 3.dat, etc.) to enable deterministic and correct recovery. CHM contains auxiliary hash map data used to perform resizing and to heuristically identify high load factors. The most crucial element of CHM data is `newTable`, a pointer to the next level of the hash table allocated. In this illustration, 1.dat was originally the top level of the table, but its values were migrated to 2.dat. Level 2.dat became the new lowest table level, discarding 1.dat from the chain. Level 3.dat is currently the top level table, and values in 2.dat are being incrementally migrated to 3.dat. Each level of the table doubles in size in this example, but table size can freely expand or compact to any multiple of 2.

C. *Lock-freedom and Resizing*

Our design achieves lock-free resizing by adapting Click's hash table [4], which uses adjacent keys and values for cache locality. Instead of grouping each key-value pair in a single atomic, each key and each value is a separate atomic. This allows for larger keys and values, since they can each be up to 62-bits, but it increases design complexity, as consistency must be maintained between the key and the value.

Our design supports all standard hash table operations as well as a non-standard `update()` operation. An `insert()` places a key-value pair into the hash map, provided that a value has not already been associated with they key. A `replace()` places a key-value pair into the hash map, overwriting any existing value. A `remove()` replaces existing values with tombstones to mark them as removed. If there is no key match in the hash map, nothing is done. An `update()` is a non-standard operation that conditionally updates a value based on the current value. This is useful, for instance, when the value in the table needs to function as a counter, such as degree counting a graph. The `update()` operation enables behaviors that, in a conventional non-blocking design without transactions, would be impossible. While `update()` has been implemented in previous works [7], our version has been optimized to use a tighter CAS loop.

PMap resizing ensures efficient table memory consumption, increasing or decreasing table size based on its load factor, and clears out tombstone values, which can accumulate to leave table slots unusable. At a broad level, PMap resizing works much like Click's hash table [4], by allocating a table with twice the capacity, then migrating each key-value pair from the old table to the new table. This approach to resizing is concurrent, allowing helping; parallel, allowing multiple threads to resize separate sections of the table; and incremental, meaning partial resizing is seen as a consistent and valid state. Unlike the state-of-the-art alternative clevel [3], this design is flexible, supporting dedicated resizing threads for efficiency or resizing directly on worker threads to ensure lock-free correctness, as well as optional strong guarantees on the number of levels maintained.

PMap migration works by placing a resize bit at each memory location as it transfers keys and values into slots at a higher table level. This bit serves to prompt other threads to help with migration. If a thread attempts to migrate a value, only to find that a value already exists in the target slot, the value being migrated is discarded, as this means the value being transferred has already been replaced. Once migration is completed for a slot, it is replaced with a migration sentinel. Once the whole table level contains migration sentinels, migration is complete. By permanently assigning keys their positions in each table level, it becomes possible to know whether a target key has been migrated from that level, if such a relocation is in progress. Ultimately, the resizing scheme reads through old levels of the table while enforcing writes to place values only in the newest table. Unlike PMap, clevel cannot reduce the size of the table because clevel inherits the design of lock-based level hashing, which uses bottom-to-top migrations to expand and top-to-bottom migrations to shrink. All threads must search and migrate in the same direction to ensure correctness, which is difficult in a non-blocking system. PMap supports reduction by making all resize operations migrate bottom-to-top.

III. Persistence

Our persistent design is an extension of the link-and-persist approach used by related works [5], [6], requiring minimum modifications to existing non-blocking data structures. So long as all atomic operations work exclusively with pointers, atomic operations can simply be replaced with link-and-persist equivalents that add the necessary flush and fence operations, leaving only recovery for developers to implement. We apply this same approach but to the values themselves, reserving a bit from the value for use with link-and-persist. Unlike alternatives [8] that require every atomic read and write to be instrumented with flush and fence operations, link-and-persist only needs to flush and fence the first time an updated variable is read or overwritten and in the worst case will persist as a factor of thread count. This makes it desirable in read-dominated workloads, which are common in graph analytics problems.

Since 64 bits is the maximum supported size of a persistent flush and our atomic operations, this is the upper limit on how big our individual in-place keys and values can be made. To track persistence, link-and-persist uses a spare bit available in 64-bit pointers to mark a dirty bit, but all 64 bits are used in 64-bit non-pointer data types. Instead, we opt to support keys and values as large as 62 bits, with one spare bit used as the dirty bit and the other for resize migrations, discussed in Section II-C. This allows the design to maintain high cache locality and eliminate key and value dereferences while using an efficient approach to limiting the number of flush and fence operations. We believe that the key and value size limitation is reasonable in our graph processing use case. For instance, degree counting will typically not need more than 62 bits to prevent integer overflow when counting edges. In cases where more than 62 bits are needed, some cache locality can be sacrificed and *pointers* to arbitrarily large objects can be stored as keys or values instead.

A. *Recovery*

To reduce the number of persists, our design does not persist auxiliary data that can be reconstructed from other persistent data, such as the size of the table. The hash table only persists the keys and values. The size and counters associated with each table can be inferred using the file size and contents while the hierarchy of levels is inferred by the filenames of the tables saved to NVM.

Link-and-persist ensures consistency between threads and NVM, and lock-freedom ensures all partial operations will not block, but correct persistence further requires these incomplete operations to be recoverable, else they become orphaned and never complete. In our design, this can only occur if the key has been inserted but the value has not. While this can be safely ignored and later discarded via resizing, we opt to explicitly mark the value as a tombstone to make it clear that the operation did not place a value.

Each level of the table is allocated as a file in persistent memory and mapped as a contiguous region of memory by `mmap`. Level files are named sequentially via a shared atomic counter to maintain table allocation order. By maintaining a strict ordering, there is no need to persist pointers to levels. Instead, table levels are opened in numeric order, with subsequent levels mapped to higher levels and linked on recovery before resuming normal execution.

IV. Correctness

The following definitions are provided to reason about durable linearizability for PMap. An execution of a concurrent system is modeled by a *history*, a finite sequence of method *invocation* and *response* events [9]. A response *matches* an invocation if they are called by the same thread on the same object. A *method call* in a history H is a pair consisting of an invocation and next matching response in H, also referred to as an *operation*. An invocation is *pending* in H if no matching response follows the invocation. An *extension* of H is a history constructed by appending responses to zero or more pending invocations of H. The notation $complete(H)$ denotes the subsequence of H consisting of all matching invocations and responses. A sequential history H is *legal* if each object subhistory is legal for that object.

Definition 1. A history H is *linearizable* if it has an extension *H'* and there is a legal sequential history S such that 1) $complete(H)$ is equivalent to S, and 2) if m_0 precedes method call m_1 in H, then the same is true in S [9].

Legal sequential history S in Definition 1 is referred to as a *linearization* of H.

Definition 2. Given an execution E, an operation O is *durable* at step t of the (extended) execution E if the following holds. For any legal execution E', which equals E in the first t steps, if the execution of the recovery of O completes in E', then for any linearization of E', O is linearized.

An operation is considered durable if there is sufficient information in NVM such that the recovery procedure causes this operation to be linearized.

Definition 3. Given an extended execution E, the *durability point* of operation O is the first point t in the execution when the operation O becomes durable.

Definition 4. Given an execution E, the durability points of the operations in the execution E imply an order on the operations, called *durability order*.

Definition 5. A linearizable object is *durably linearizable* if for all executions E of the object, 1) the durability point of each operation is between its invocation and response, and 2) there exists a linearization of E whose order of operations is the same as the durability order of operations in E [10].

A. Durable Linearizability

To prove that the PMap is durably linearizable, it must be shown that for all multithreaded executions E, 1) the durability point of each operation is between its invocation and response, and 2) there exists a linearization of E whose order of operations is the same as the durability order of operations in E.

Theorem 1. *The PMap is durably linearizable.*

Proof. First it is shown that the durability point of each operation is between its invocation and response. According to the link-and-persist technique [5], [6] discussed in Section III, all atomic words set the "dirty bit" prior to being written by CAS. After the atomic word is written by CAS, the operation persists the atomic word using a flush and fence, then atomically removes the mark. The durability point for `insert` occurs when the insert value is persisted after the CAS successfully inserts the new value into a slot with a tombstone. The durability point for `remove` occurs when the tombstone is persisted after the CAS successfully replaces the old value with a tombstone. The durability point for `replace` and `update` occurs when the new value is persisted after the CAS successfully updates the old value to the new value.

Next it is shown that there exists a linearization of E whose order of operations is the same as the durability order of operations in E. As previously mentioned, a successful CAS always updates an atomic word at a memory location with the dirty bit set. If any operation attempts to read or write an atomic word before it is persisted, it will observe that the dirty bit is set and help persist the atomic word prior to proceeding with its own operation. Since the persist order is equivalent to the CAS order for a particular memory location and updates by CAS to different memory locations are commutative, there exists a linearization of E whose order of operations is the same as the durability order of operations in E.

If a crash occurs, the recovery procedure is invoked by the main thread to restore the state of the PMap. It now must be shown that the restored state reflects a linearization of E whose order of operations is the same as the durability order of the operations in E. Let op' be an operation that has persisted the key but not yet persisted the value. Let $op_1, op_2, ..., op_{n-1}, op_n, op'$ be the history of operations involving an arbitrary memory location up to a crash event. All operations from op_1 to op_n are guaranteed to be durable linearizable because the link-and-persist technique requires that an operation that accesses an atomic word with the dirty bit set must help complete that operation before proceeding with its own operation. When the recovery procedure is invoked, any partial operations are identified and reverted. Therefore, the history of operations to any arbitrary memory location becomes $op_1, op_2, ..., op_{n-1}, op_n$. When considering the composition of operations to all memory locations, the operations can be rearranged to form a linearization of E whose order of operations is the same as the durability order of operations in E since operations to different memory locations are commutative. □

V. Experimental Results

A. Experimental Setup

Our experiments are configured to run on a Linux server configured with 134GB of DRAM and 248GB of Intel Optane DC provisioned in App Direct mode and mounted in Filesystem-DAX (fsdax) mode. PMap interfaces with this filesystem using `mmap()`, flushing with `clflush` and fencing with `sfence`. Two Intel Xeon Gold 6230 CPUs are set up for NUMA. Each CPU has 20 cores and 40 threads for a total of 80 threads. All code was written in C++ and built in GCC using `-O3`, `-march=native`, and `-flto` flags.

We ran tests on the following data structures:

- **PMap:** PMap is our new design proposed in this paper. It is based on the Click hash map [4], offering asynchronous parallel resizing and lock-freedom, but with additional persistence guarantees added. It uses open addressing with linear probing to improve temporal locality and reduce dereferencing.
- **clevel:** Concurrent level hashing is a lock-free implementation of conventional lock-based level hashing [3]. This design provides asynchronous resizing, lock-freedom, and open addressing in NVM, making it the most closely related work and the current state-of-the-art.
- **OneFile:** OneFile provides a library for wait-free persistent transactions [11]. We use the hash map included in the OneFile code provided by the authors in our evaluation. This design is node-based rather than using open addressing.
- **STL:** The std::map is provided by C++ as a part of the standard template library. This implementation uses a global lock to provide thread safety. It is the only design in our performance testing that does not provide persistence in NVM, instead working as a DRAM baseline to compare against the impact of NVM and the flush and fence constraints used by the alternatives.
- **PMDK:** The concurrent_hash_map is provided by the libpmem library as a convenient and readily available NVM hash map [12]. It is based on the Intel TBB hash map [13] but modified to support NVM. It uses buckets restricted by reader-writer locks.

All tests use 62-bit words to prevent the need for pointer dereferencing in our design. Our tested design does not assign dedicated resizing threads, devoting all active threads as worker threads. Threads perform whole-table resizes when a resize is invoked, to limit the number of active tables. To provide fair testing, our open addressing approaches, PMap and clevel, both use an initial table size of 2^{14}. While fully migrated tables can be safely deleted, our preliminary code offers no garbage collection. Levels can be tracked and reclaimed using traditional lock-free approaches, including reference counting, hazard pointers, or epoch-based reclamation. These systems do not need persistence in our design to function properly, as we have implemented comprehensive garbage collection for tables on recovery, inferred based on table contents. Garbage collection was avoided to benchmark the data structure itself, without the overheads of the memory reclamation system.

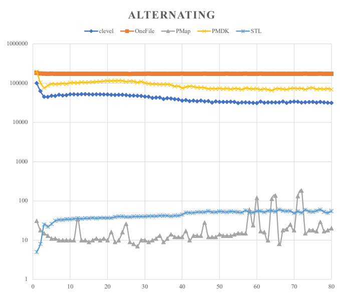
We ran the following tests to evaluate performance:

- **Alternating:** This test pre-fills the table to 50% capacity, then each thread alternates between inserting and removing key-value pairs. Each key is unique to the running thread to reduce contention. After performing 400,000 operations in our tests, the overall run time is measured at various thread counts. Results are illustrated in Fig. 2a. In this test, we find that PMap averages 3122x faster than the state-of-the-art, clevel.
- **Random:** This test pre-fills the table to 50% capacity, then performs all operations at random with equal probability. Some operations are substituted when not natively supported by the tested structure. For instance, `increment()` is not supported by clevel, so `insert()` is used instead. After performing 400,000 operations in our tests, the overall run time is measured at various thread counts. Results are illustrated in Fig. 2b. In this test, we find that PMap averages 334x faster than the state-of-the-art, clevel.
- **Degree testing:** R-MAT [14] is a graph generating model. It lists node connections as pairs. This test assigns up to four threads separate RMAT files to parse into the hash map. It works by counting the number of connections leaving each node using an `increment()`-based `update()` function. This is meant to simulate the behavior of degree counting, a common graph processing task. While PMap and STL easily support increment, the alternatives use insert as a basic, albeit insufficient substitute in these tests. Results are illustrated in Fig. 2c. In this test, we find that PMap averages 6551x faster than the state-of-the-art, clevel.

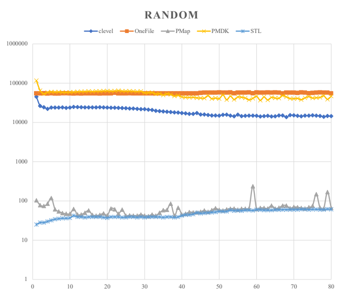
B. Discussion

In all testing conducted, PMap outperforms all NVM alternatives. We credit these results to the simplified memory allocation used, where whole tables are allocated rather than nodes or keys and values. We also believe that placing keys and values directly in the table, unlike the NVM alternatives, provides PMap with a performance advantage via excellent cache locality and a reduction in dereferences.

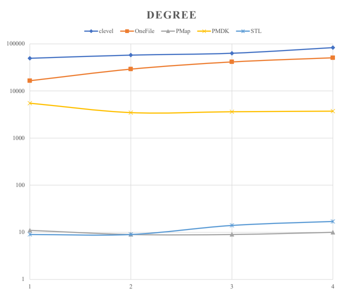
Clevel offers the previous state-of-the-art performance. Its design is similar to ours, but it requires keys and values be placed atomically via pointer, with the resulting dereferences



(a) The alternating test, comparing our design with state-of-the-art alternatives.



(b) A random operation test, comparing our design with state-of-the-art alternatives.



(c) A node degree test, comparing our design with state-of-the-art alternatives.

Fig. 2: Performance tests: Comparisons of our design (PMap) against state-of-the-art alternatives. In each sub-figure, the X-axis is the thread count, and the Y-axis is the time to complete the test in milliseconds.

reducing performance. Since keys and values are held via separate objects, additional overhead is introduced by performing memory allocation in NVM.

The other designs have limitations as well. OneFile offers relatively poor performance because its design is meant to be general and wait-free, rather than efficient. Wait-freedom and full transaction support are stronger progress and isolation guarantees than are required to implement a practical NVM hash map design. PMDK is designed using straightforward locks, limiting its performance. STL performs well because it uses DRAM instead of slower NVM memory and uses no flush or fence operations. This improves performance at the cost of persistence.

VI. Related Work

Hash tables are fundamental and thus have a large existing body of research work describing concurrent, resizable, and non-volatile designs. Early concurrent hash maps either did not provide resizing [15], [16] or provided slow resizing [17], [18]. Newer works include an HTM-based cuckoo hash table [19] and a non-blocking phase-concurrent hash table [20].

Several hash tables designs have focused on fast resizing using a variety of approaches. Novel techniques include a linked-list-based hash table that can split and join buckets dynamically [21], wait-free resizing via extendable buckets [7], [22], parallel resizing using a hash function that places keys in the same region regardless of table size [23], and lock-free resizing with open addressing [4], [24].

A. Non-volatile Hash Maps

PMDK is an NVM framework developed by Intel for use with Optane DC [12]. It provides libraries for developing NVM applications, including NVM data structures. For our purposes, the PMDK concurrent hash map offers a baseline implementation for comparison.

OneFile from Ramalhete et al. is the first NVM framework to guarantee wait-free transactions in NVM [11]. It values general applicability of correctness conditions for ease of use over performance. Its full transactional guarantees are useful but nonessential for our design.

Alternative persistent hash tables include Dalí, a periodic persistence approach where updates are persisted using a global fence [25], log-free hashing with resizing [26], [27], and general-purpose transformations such as the Pronto library [28], phase change memory (PCM) [29], Mnemosyne [30], and iDO [31]. More recent designs explicitly accommodate Optane DC's limited bandwidth, including Dash [32], which uses compressed fingerprints of entries for read-reduced probing and Rewo-Hash [33], which synchronizes a volatile copy of the persistent table in DRAM for fast searches. Concurrent level hashing (clevel) was the first concurrent lock-free hash table with open addressing developed for use with non-volatile memory [3]. It builds on level hashing [26], [27] for the hash table, link-and-persist [5], [6] for persistence, and offers a novel lock-free design.

VII. Conclusions

NVM presents unique challenges for in-memory data structures. Our work developed and tested a non-volatile, scalable hash map, applicable in large graph processing workloads while performing well enough to be used in a wide variety of tasks. PMap is currently the best performing hash map to offer lock-freedom and open addressing in NVM. Our performance testing finds this design offers significant performance gains over the state of the art, averaging 3122x faster than the previous state-of-the-art, clevel, while maintaining lock-free progress and persistent recovery. The source code for this work is available for evaluation at https://github.com/ucf-cs/PMap.

PMap, as written, is limited to keys and values as large as 62 bits each. This is a limitation of the implementation rather than the design. It can be extended to store arbitrarily large elements by storing pointers instead of keys and values. Concurrent level hashing uses this approach with optimizations to reduce dereferences [3]. Properly implementing this requires a mechanism to persist and recover external key and value objects, which is beyond the scope of this paper. Additionally, this extension would require pointer dereferences to access keys and values, reducing cache locality and thus performance.

Using linear probing results in a more straightforward design and improved cache locality when probing, but it also results in a poor performance at high load factors. We believe this is a worthwhile tradeoff between performance and functionality, offered in exchange for space, but future work should examine different designs to improve high load factor performance. Cuckoo hashing or Hopscotch hashing would greatly improve load factor performance. This requires relocating keys, which violates the constraints required to support our resizing scheme and results in enormous overhead to accomplish via lock-free KCAS [34], [35]. Using multiple hash functions to place keys may also be possible, but this increases the complexity and overhead of resizing, as multiple locations must be checked for each key to verify migration.

Acknowledgment

We thank Dr. Yan Solihin and Derrick Greenspan in the ARPERS research group for granting us access to their Optane DC machine, which was instrumental in conducting our PMEM performance tests. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL-LDRD 20-FS-040. LLNL-CONF-818404-DRAFT.

References

[1] R. Pearce and M. Gokhale, "Persistent memory evaluation and experiments," in *Persistent Programming In Real Life (PIRL) 2019*, Jul. 2019. [Online]. Available: https://pirl.nvsl.io/PIRL2019-content/PIRL-2019-Maya-Gokahle.pdf

[2] R. Pearce, T. Steil, B. W. Priest, and G. Sanders, "One quadrillion triangles queried on one million processors," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–5.

[3] Z. Chen, Y. Huang, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 799–812. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/chen

[4] C. Click, "A lock-free wait-free hash table," *work presented as invited speaker at Stanford*, 2008.

[5] 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)*. IEEE, 2018, pp. 461–472.

[6] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 373–386. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/david

[7] P. Laborde, S. Feldman, and D. Dechev, "A wait-free hash map," *International Journal of Parallel Programming*, vol. 45, no. 3, pp. 421–448, 2017.

[8] J. Izraelevitz, H. Mendes, and M. L. Scott, "Brief announcement: Preserving happens-before in persistent memory," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016, pp. 157–159.

[9] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[10] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," *SIGPLAN Not.*, vol. 53, no. 1, pp. 28–40, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3200691.3178490

[11] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, "Onefile: A wait-free persistent transactional memory," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 151–163.

[12] I. Corporation, "pmem.io persistent memory programming," 2014 (Accessed September 10, 2020), https://pmem.io/.

[13] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, p. 298, Apr. 2008.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[15] C. Purcell and T. Harris, "Non-blocking hashtables with open addressing," in *International Symposium on Distributed Computing*. Springer, 2005, pp. 108–121.

[16] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *Journal of Parallel and Distributed Computing*, vol. 70, no. 8, pp. 839–848, 2010.

[17] H. Gao, J. F. Groote, and W. H. Hesselink, "Efficient almost wait-free parallel accessible dynamic hashtables," *Technical Report CS-Report 03-03*, 2003.

[18] H. Gao, J. F. Groote, and W. H. Hesselink, "Lock-free dynamic hash tables with open addressing," *Distributed Computing*, vol. 18, no. 1, pp. 21–42, 2005.

[19] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[20] J. Shun and G. E. Blelloch, "Phase-concurrent hash tables for determinism," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, 2014, pp. 96–107.

[21] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM (JACM)*, vol. 53, no. 3, pp. 379–405, 2006.

[22] P. Fatourou, N. D. Kallimanis, and T. Ropars, "An efficient wait-free resizable hash table," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 111–120.

[23] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general (?)!" *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 4, pp. 1–32, 2019.

[24] A. Malakhov, "Per-bucket concurrent rehashing algorithms," *arXiv preprint arXiv:1509.02235*, 2015.

[25] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[26] 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)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 461–476. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo

[27] P. Zuo, Y. Hua, and J. Wu, "Level hashing: A high-performance and flexible-resizing persistent hashing index structure," *ACM Transactions on Storage (TOS)*, vol. 15, no. 2, pp. 1–30, 2019.

[28] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 789–806.

[29] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2016.

[30] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[31] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 258–270.

[32] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 10, p. 1147–1161, Apr. 2020. [Online]. Available: https://doi-org.ezproxy.net.ucf.edu/10.14778/3389133.3389134

[33] K. Huang, Y. Yan, and L. Huang, "Revisiting persistent hash table design for commercial non-volatile memory," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 708–713.

[34] N. Nguyen and P. Tsigas, "Lock-free cuckoo hashing," in *2014 IEEE 34th international conference on distributed computing systems*. IEEE, 2014, pp. 627–636.

[35] R. Kelly, B. A. Pearlmutter, and P. Maguire, "Lock-free hopscotch hashing," *arXiv preprint arXiv:1911.03028*, 2019.