Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs

Yihe Huang, *Harvard University;* Matej Pavlovic, *EPFL;*
Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan, *Oracle Labs*

https://www.usenix.org/conference/atc18/presentation/huang

ISBN 978-1-939133-02-1

Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs

Yihe Huang[‡] *Matej Pavlovic*[†*] *Virendra J. Marathe*[◇] *Margo Seltzer*[◇‡]
Tim Harris[◇] *Steve Byan*[◇]
Harvard University[‡] *EPFL*[†] *Oracle Labs*[◇]

Abstract

Key-Value (K-V) stores are an integral building block of modern datacenter applications. With byte-addressable persistent memory (PM) technologies, such as Intel/Micron's 3D XPoint, on the horizon, there has been an influx of new high performance K-V stores that leverage PM for performance. However, there remains a significant performance gap between PM optimized K-V stores and DRAM resident ones, largely reflecting the gap between projected PM latency relative to that of DRAM. We address that performance gap with Bullet, a K-V store that leverages both the byte-addressability of PM and the lower latency of DRAM, using a technique called *cross-referencing logs (CRLs)* to keep most PM updates off the critical path. Bullet delivers performance approaching that of DRAM resident K-V stores by maintaining two hash tables, one in the slower (backend) PM and the other in the faster (frontend) DRAM. CRLs are a scalable persistent logging mechanism that keeps the two copies mutually consistent. Bullet also incorporates several critical optimizations, such as dynamic load balancing between frontend and backend threads, support for nonblocking `Gets`, and opportunistic omission of stale updates in the backend. This combination of implementation techniques delivers performance within 5% of that of DRAM-only key-value stores for realistic (read-heavy) workloads. Our general approach, based on CRLs, is "universal" in that it can be used to turn any volatile K-V store into a persistent one (or vice-versa, provide a fast cache for a persistent K-V store).

1 Introduction

Key-value (K-V) stores with simple `Get`/`Put` based interfaces have become an integral part of modern data center infrastructures. The list of successfully deployed K-V stores is long – Cassandra [28], Dynamo [13], LevelDB [30], Memcached [36], Redis [44], Swift [48] – to name just a few. The research community continues to publish K-V store improvements along a variety of dimensions including network stack optimizations, cache management, improved parallelism, hardware extensions, etc. [5, 14, 15, 20, 27, 31, 33, 32, 34, 37, 40, 51, 53, 56]. However, many of these works assume that the K-V store is a volatile cache for a backend database. Most of the persistent K-V stores [7, 13, 18, 28, 30, 34, 51, 44, 48] assume a slow, block-based storage medium, and therefore, marshal updates into blocks written to the file system.

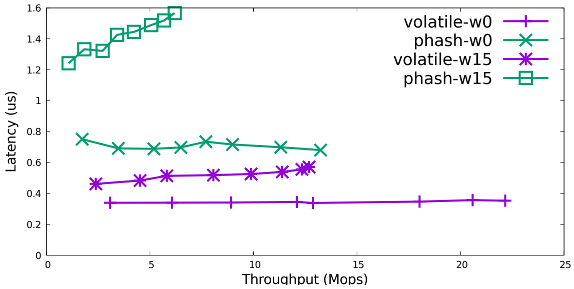


Figure 1: Throughput vs. Latency results of hash table based K-V stores: (i) phash, hosted entirely in emulated PM (Intel's Software Emulation Platform [43, 57]), and (ii) an almost identical K-V store hosted entirely in DRAM (volatile). The emulated PM has 300 nanosecond load latency and bandwidth identical to that of DRAM; DRAM latency is approximately 150 nanoseconds. 0 and 15 represent the percent of K-V accesses that are `Puts`; keys are selected according to a zipfian distribution. The points on the curves represent the number of threads used in the tests, ranging from 2 to 16 in increments of 2.

At the same time, byte-addressable persistent memory technologies are emerging, e.g., *spin-transfer torque MRAM (STT-MRAM)* [21, 23], *memristors* [46]), and most notably, the Intel/Micron 3D XPoint persistent memory [1]. These technologies provide the persistence of traditional storage media (SSDs, HDDs) with the byte addressability and performance approaching that of DRAM (100-1000x faster than state-of-the-art NAND flash). Byte addressability allows `load/store` access to persistence (as opposed to the traditional file system interface). As a result, these technologies can profoundly change how we manage persistent data.

The research community has recognized this potential, producing an endless stream of new, PM-optimized K-V stores that leverage PM's byte addressability and low latency, yielding systems that greatly outperform traditional block-based approaches [3, 8, 9, 12, 22, 39, 41, 54, 55, 58]. While this body of work has grown rapidly, most of it ignores the fact that for the forseeable future, PM will be much slower than DRAM [47], making PM resident K-V stores significantly slower than their DRAM counterparts. Figure 1 illustrates the performance gap between K-V stores hosted in DRAM and *emulated* PM. Their implementations differ only in their failure semantics (section 3) and pointer representation (section 6). The 0% writes curves in the graph mirror the $2X$ latency gap between DRAM and emulated PM. This $2X$ gap grows to $3-4.5X$ in the 15%-write case, since writes use expensive persist barriers and transactions for failure atomic updates to the persistent data structures.

Recent PM-based K-V store proposals [41, 54, 55] address this problem by partitioning their data structures between faster DRAM and slower PM, with the DRAM resident structures reconstructed during recovery/warmup. However, these optimizations focus exclusively on B-Tree based indexing structures, not on hash table based structures, which are predominantly used in workloads with `Get`/`Put` point queries. Since these hash tables are central to many popular K-V stores [30, 36, 44], leveraging both DRAM and PM in their implementations is critical to their performance.

We present Bullet, a new K-V store designed for multi-/many-core systems equipped with persistent memory. Bullet explicitly leverages the combination of fast DRAM and slower, byte-addressable PM, to deliver performance comparable to that of a DRAM resident K-V store in realistic workloads. Bullet's architecture is designed to handle most, if not all, client requests in the faster DRAM, minimizing the number of PM accesses on the critical path. This naturally leads to an architecture with a DRAM resident cache, similar to the approach taken by traditional databases and K-V stores. However, Bullet deviates from traditional approaches in that the cached *frontend* hash table and the persistent *backend* hash table representations are virtually identical – differing only in their pointer representations (section 6) and failure handling semantics. This facilitates efficient access to backend data whenever there is a miss in the frontend – PM's byte addressability plays a critical role in making this possible.

We keep the frontend and backend mutually consistent by employing a novel, efficient, and highly concurrent logging scheme, called *cross-referencing logs (CRLs)*. In an architecture using per-thread persistent logs, CRLs track ordering dependencies between log records using simple cross-log links instead of synchronizing the threads' log access [29, 52]. Bullet processes `Get` requests exclusively in the frontend, without log access. On their critical path, `Put` requests access the frontend as well, while also writing log records to CRLs. This results in a single thread-local log append per update.

Backend threads, called *log gleaners*, apply persisted log records to the backend hash table. We use an *epoch* based scheme to apply log records to the backend in batches. The epoch based scheme's primary purpose is to enable correct log space reclamation. The backend's hash table updates must be applied in a crash consistent manner. We address this problem using a backend runtime system [35] that supports *failure atomic transactions* similar to several other persistent memory transaction runtimes [16, 50]. The resulting code path is complex, but *not on the critical path* of client requests.

We apply four key optimizations in Bullet: 1) fully decoupling frontend execution from PM performance on `Put` operations, 2) nonblocking `Gets`, 3) dynamic thread switching between the frontend and backend, based on the write-load in the system, and 4) opportunistic `Put` collapsing. Our base design, coupled with these optimizations, make Bullet's performance close to that of a DRAM resident K-V store: For realistic, read-heavy workloads, Bullet either matches or comes close to the performance of a DRAM-resident volatile K-V store, delivering throughput and latency $2X$ better than that of a state-of-the-art hash table based K-V store, HiKV [54], on a system with emulated PM whose access latency is $2X$ of DRAM access latency. For pathological write-heavy workloads, Bullet's throughput is comparable to or better than that of HiKV and its operations' latency is approximately $25-50\%$ lower. Relative to a volatile K-V store, Bullet's latency and throughput degrade by approximately 50% under write-heavy workloads.

2 Bullet's Architecture

2.1 Overview

Figure 2 depicts the high level architecture of Bullet, separated into the frontend and backend components, each of which contains almost identical hash tables. The frontend resides in the *volatile domain* (DRAM). It contains a configurable number of threads that process incoming requests, applying them to its hash table. Each frontend thread additionally "passes on" update requests to the backend, by appending update requests to a *thread-local* persistent log. An update completes when it has been safely written to the log. The backend resides in the *persistent domain* (PM). The backend's *log gleaner* threads periodically read requests from their corresponding persistent logs and apply them to the persistent hash table, in a failure-atomic and correctly ordered manner. In this "base" configuration, each persistent log maps both to a *log writer* thread in the frontend and a log gleaner thread in the backend.

While processing client requests, a frontend thread first looks up the target key in the frontend K-V store. If the lookup succeeds, the frontend applies the operation. If it is an update (`Put` or `Remove`), the thread also appends the `<opcode,payload>` tuple to its persistent log. If the lookup in the frontend fails, the thread issues a lookup to the backend. A successful lookup creates a copy of the key-value pair in the frontend, at which point the operation proceeds as if the original frontend lookup succeeded. If the lookup fails: (i) a `Get` returns a failure

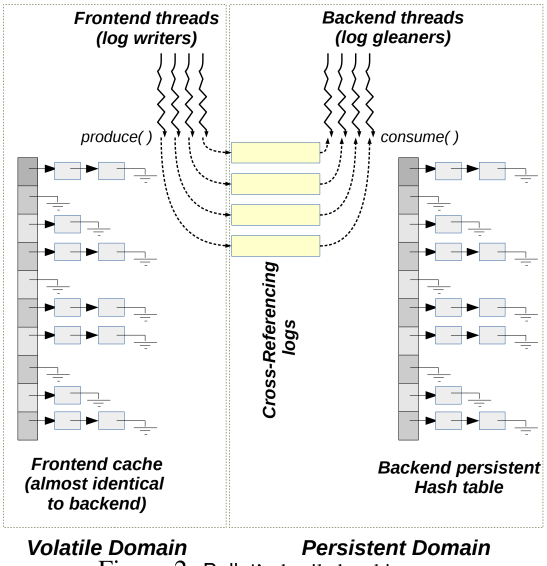


Figure 2: Bullet's detailed architecture.

code to the client, (ii) a `Put` inserts the pair into the frontend, including the log write, and (iii) a `Remove` returns with a failure code.

The rest of this section details our solutions to a number of technical challenges: persistent and volatile hash table implementation (subsection 2.2), the parallel logging scheme (subsection 2.3), correct coordination between frontend and backend threads (subsection 2.4), and failure atomic updates (section 3).

2.2 Hash Tables

As shown in Figure 2, Bullet's frontend hash table is in DRAM and therefore volatile. It supports the standard K-V operations: `Get`, `Put`, and `Remove`. The hash table is similar in structure to other key-value stores [36]: It is closed addressed, with chaining to handle hash conflicts. It grows via a background thread responsible for doubling the number of hash table buckets when occupancy crosses a threshold size (twice the number of buckets). Regular operations can occur concurrently with growing the table. Each hash table bucket has its own reader-writer spinlock for thread synchronization – lookups acquire the spinlocks for reading (shared), and updates acquire the spinlocks for writing (exclusive).

The backend hash table is structurally identical to the frontend one, with its own per-bucket chains and spinlocks. However, unlike the frontend (volatile) hash table, the backend hash table resides in persistent memory and must survive failures. Bullet uses failure atomic transactions for `Put` and `Remove` operations to provide this guarantee (section 3). `Get`s execute identically to those in the frontend (except a failure to find a key is always a failure in the backend, while the frontend has to check the backend before failing).

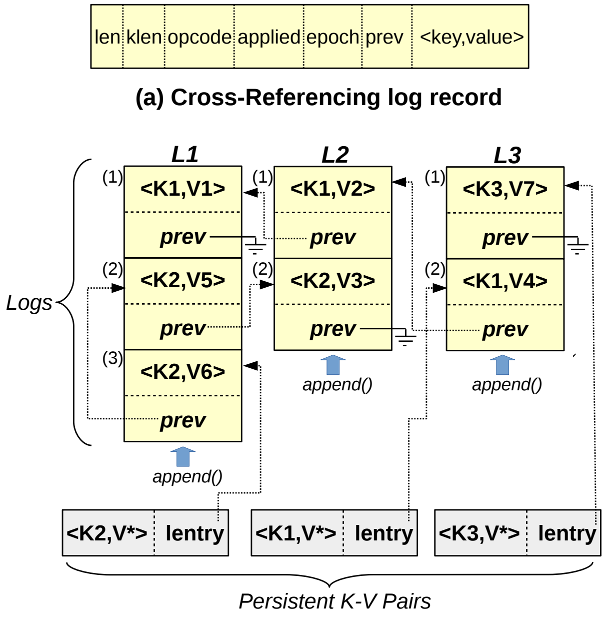
The per-bucket spinlocks in the persistent hash table are used only for synchronization between concurrent backend threads and are semantically volatile. We found placing the spinlocks in the bucket extremely convenient, with the added benefit of improved cache locality compared to an alternative where the spinlocks are mapped elsewhere in DRAM. Since a bucket's spinlock resides in persistent memory, its state can persist at arbitrary times (e.g., due to cache line evictions). A failure could leave a spinlock in the *locked* state. We leverage a generation number technique [10] to reinitialize such locks after a restart – Bullet increments a global persistent generation number during every warm-up and compares that generation number to a generation number contained in every lock. If the generation numbers do not match, Bullet treats the lock as available and reinitializes it.

2.3 Cross-Referencing Logs

The frontend communicates updates to the backend via a log. In a conventional, centralized log design [25, 26, 38], the log becomes a bottleneck, because concurrent updates must all append records to the log. Thread-local logs neatly address this contention problem, but introduce a new challenge: records from a multitude of logs must be applied to the backend in the correct order – the order in which the corresponding operations were applied in the frontend. While prior systems partition the key space so that all updates to a particular K-V pair appear in the same log file (e.g., [34]), Bullet does not partition the data and K-V pair updates can happen in any thread. This way Bullet is not susceptible to load balancing issues encountered in partitioned K-V stores [34]. We address the ordering problem in a different way: We introduce *cross-referencing logs (CRLs)*, to provide highly scalable, concurrent logging on PM, without relying on centralized clocks [29, 52] to enforce a total order of update operations.

Figure 3 illustrates CRLs. Each frontend log writer thread maintains its own persistent log. Logically, each log record is a `<opcode,payload>` tuple. Opcode allows the application to define high-level operations expressed by each log record. For example, when Bullet manages a hash table of lists, each list append can be expressed by a single log record, where the opcode refers to the list append operation, and the payload contains the record identifier (a reference to the list in question) plus the value to be appended. The order in which non-commutative operations like this are applied is important, hence the necessity of the CRL scheme. The logs require no synchronization on appends, because there is only one writer per log. The backend maintains corresponding log gleaner threads that consume log records and apply them to the backend persistent hash table in a failure-atomic manner.

The logs are structured so that log gleaners can easily



(a) Cross-Referencing log record

(b) Cross-Referencing log structure

Figure 3: Cross-Referencing Log (CRL) architecture.

determine the correct order in which to apply log records. Figure 3a shows the log record layout. The `len`, `klen`, `opcode`, `<key,value>` fields contain the information implied by their names. The `applied` field contains a flag indicating whether the backend K-V store accurately reflects the log record. The `prev` field contains a persistent pointer to the prior log record, if one exists, for the given key. We defer discussion of `epoch` until subsection 2.4.

Appending Log Records: Figure 3b depicts three logs $L1$, $L2$, and $L3$ containing log records for keys $K1$, $K2$, and $K3$. The `lentry` field of the persistent key-value pairs (shown at the bottom) contains a persistent pointer to the most recent log record for the key-value pair. Thus, the list formed by the `lentry` and `prev` pointers represents the evolution of a key-value pair in reverse chronological order, where the log record containing a NULL `prev` pointer indicates the first update to the pair present in any of the logs. The list for a specific key can *criss-cross* among multiple logs, hence the name *cross-referencing logs*. For instance, log records for key $K1$ appear in all three logs, whereas log records for key $K2$ appear only in logs $L1$ and $L2$.

Before a log writer appends a log record, it acquires the key's hash bucket lock, to ensure that it is the only writer for the target key-value pair. Then, the writer (i) populates the log record at the tail end of the log, setting the log record's `prev` field to the value stored in the K-V pair's `lentry`, (ii) persists the log record, (iii) updates and persists the log's tail index, and finally (iv) updates and persists the key-value pair's `lentry` pointer, thus completing the linked list insertion. In all, an append requires 3 persist barriers.

Applying Log Records: Gleaner threads periodically scan logs and apply log records to the backend hash table in a failure-atomic manner. A log gleaner starts processing from the beginning of the log (the head). For each log record encountered, the gleaner looks up the corresponding key-value pair in the backend persistent hash table; a new key-value pair is created if necessary.

The gleaner retrieves the key-value pair's `lentry` to process all existing log records for that key-value pair. At this point, we need to ensure that at most one gleaner is processing log records for a given key-value pair. To that end, we add another spinlock that enables only one gleaner to apply all the log records for a key-value pair. This spinlock is placed in the key-value pair itself. A gleaner must acquire this spinlock before processing the log records for the key-value pair. The gleaner then traverses to the end of the list, checking the `applied` flag of each log record to determine the point from which the gleaner needs to apply log records. Upon finding the last (chronologically the first) unapplied log record, the gleaner applies the log records in the chronological order determined by the linked list (i.e., in the reverse order of the list). The gleaner sets the `applied` field after applying the log record to the persistent hash table. We discuss the transaction mechanism that ensures recoverability of these updates in section 3.

After applying all the log records for a key-value pair, the gleaner can reset `lentry` to `NULL`. This however races with a frontend log writer's append for the same key-value pair, which requires an update to the key-value pair's `lentry`. Fortunately the data race can be avoided using a `compare-and-swap` instruction, by both the appender and the gleaner, to atomically change `lentry`.

Consider the example in Figure 3. A gleaner for log $L1$ will first encounter log record labeled (1). It uses the log record's key, $K1$, to retrieve the corresponding persistent key-value pair (at the bottom of Figure 3b). From that key-value pair object, the gleaner begins at the end of the log record list at $L3(2)$, then continues to $L2(1)$ and finally $L1(1)$. It then applies each of those log records in reverse traversal order.

Handling Removes: Removes are unique, in that they logically require removing a record at the front end, but the same record at the persistent back end may not be removed at the same time due to log delays. If a deletion is followed by a re-insertion of the same key, the front end and back end can grow inconsistent, due to the fact that CRL relies on the back end record to generate cross-references. To address this problem, we keep the front end record alive as long as we need to by using a special "tombstone" marker. Appending a delete log record only sets the tombstone marker at the front end, but does not remove the record. Future look-ups on the front end regarding this record now return "not found", until a reinsertion clears this tombstone marker. The front end records marked with tombstones are only physically removed when the corresponding records in the back end are removed during the log gleaning phase.

Rationale: While cross-referencing logs are interesting, one could argue that the criss-crossing could lead to bad cache locality for log gleaner threads. However, it is a trade-off – a thread may suffer poorer locality in its log traversal, but it enjoys superior cache locality, by repeatedly acting upon the same key-value pair. This cache benefit is further enhanced, because log records are concise representations of operations, but the operations themselves tend to lead to "write amplification", accessing and updating many more memory locations than a single log entry. By continuing to operate on the same key-value pair, we observe that those accesses are far more likely to produce cache hits. Additionally, gleaners never block behind other gleaners. If a gleaner detects that the key-value pair it needs to process is already locked by another gleaner, it can safely assume that the spinlock owner will apply the log record. As a result, the gleaner simply skips that log record. This approach works for the *fail-stop* failure model we assume – a failure terminates the entire key-value store process.

2.4 Log Space Reclamation

The cross-referencing logs that act as bridges between Bullet's frontend and backend do not grow indefinitely. In fact, they are circular logs and contain persistent `head` and `tail` indexes. To keep the system running without interrupt, Bullet must recycle log space.

The log gleaners work in phases or *epochs*. Between epochs, the gleaners wait for a signal from the *epoch advancer* thread, which periodically tells the gleaners to start applying logs records. Each gleaner reads the log, beginning at the head, and applies the log records as described above. However, it does not advance its log's head index. Instead, the epoch advancer periodically terminates the current epoch by telling the gleaners to stop processing the log. At this point, the epoch advancer updates each gleaners' head index. If a log writer fills the log more quickly than the corresponding log gleaner applies the log, the log can fill. If this happens, the writer blocks until the gleaner frees space in the log.

3 Failure Atomic Transactions

To ensure a consistent state after system failure, the backend's hash table updates must be failure atomic. We use failure atomic persistent memory transactions. Similar to prior work [6, 10, 16, 50], we developed a persistent memory *access library* [35], which contains support for low level programming abstractions that greatly simplify application development for persistent memory. Our access library supports transactions that provide failure atomicity guarantees for updates to persistent memory.

```
pm_txn_t *txn_begin();
txn_state_t txn_commit(txn);
void txn_read(txn, src, len, dst);
void txn_write(txn, dst, len, src);
... // other accessor functions
pm_region_t *pm_region_open(path);
void pm_region_close(region;
void *pm_get_region_root(region);
void pm_set_region_root(region,addr);
... // other region management functions
void *pm_alloc(txn, len);
void pm_free(txn, addr);
```

Figure 4: Base persistent transactions API.

Figure 4 presents our transaction runtime's API. The interface provides `txn_begin` and `txn_commit` functions to delineate transaction boundaries and various `txn_read` and `txn_write` accessor functions for transactional reads and writes of persistent data. The interface also provides transactional variants of general purpose `libc` functionality, such as `memcpy`, `memset`, `memcmp`, etc. We provide "flat nesting" semantics [19]. The transaction mechanism provides *only* failure atomicity semantics; it does not transparently manage concurrency control, as do some software transactional memory runtimes [10, 50]. Bullet itself performs the necessary synchronization to avoid data races and deadlocks.

The access library also provides a *persistent region* abstraction [6, 10, 50]. The persistent region builds over the `mmap` interface, mapping a persistent memory file into the application's address space [49]. The persistent region contains a persistent heap, modeled after the Hoard allocator [4, 50]. Application data hosted in a persistent region can be made reachable via a special, per region, *root* pointer. Bullet uses the region's root pointer to reach its persistent hash table and cross-referencing logs. Finally, the access library uses redo logging [16, 35, 50] to implement failure atomic writes.

4 Optimizations

4.1 Tightening the Update Critical Path

Bullet is designed to streamline critical paths of update operations. To that end, Bullet moves the persistent hash table's failure-atomic updates off the critical path. However, the design presented thus far does not entirely remove transactions from the update critical path. On a `Put` operation, if the key does not exist in either the frontend or backend hash tables, Bullet allocates a new *persistent* K-V pair object, storing a reference to it in the log record. Furthermore, when the persistent log append completes, we must also update the the key-value pair's `lentry` to reference that newly created log record. Accessing the persistent K-V pair itself requires a lookup in the backend hash table, which is costly due to the relatively slower persistent memory. All these accesses and updates contribute significant latency to the frontend update operations.

We address this problem by completely decoupling backend data accesses from the frontend update operations, by moving the `lentry` pointer to the frontend hash table's K-V pair. This gets rid of the requirement to locate, and possibly allocate, the backend's K-V pair for a new key. It also eliminates the expensive persist barrier required to persist the `lentry`, since it is no longer persistent; it's part of the volatile copy of the K-V pair. This also eliminates the need for transactions in the frontend, thereby considerably shortening the frontend's update critical path.

4.2 Nonblocking `Gets`

Bullet's "base" version, as described in section 2, uses reader-writer locks to synchronize access to the frontend and backend buckets. While these work well with few frontend and backend threads, they do lead to increased cache contention between concurrent readers on the lock's readers counter – the lock implementation uses a signed integer, where a value greater than 0 indicates one or more readers, and a -1 indicates a writer. The resulting cache contention can restrict scalability. This can be especially pronounced in workloads where accesses follow a power-law distribution and are skewed to a small set of K-V pairs, as is experienced by real world K-V stores [15, 40].

As in prior work [15], we support nonblocking `Get` operations. The principal hurdle for nonblocking `Gets` is memory reclamation – a `Put` or `Remove` can deallocate an object being read by a concurrent `Get`. We need support to *lazily* reclaim the removed objects. Bullet's epochs neatly enable this lazy memory reclamation. The epoch advancer thread periodically increments Bullet's global epoch number. Each frontend thread maintains a local epoch equal to the global epoch number at the beginning of an operation.

When freeing an object, the frontend thread enqueues the object on its local free queue. The enqueued node contains a pointer to the object and the thread's epoch number. On each enqueue, the frontend thread frees the head node of the queue if its epoch is older than the smallest epoch of all the frontend workers. The smallest epoch is a conservative approximation of workers' epochs – it is computed periodically by the epoch advancer thread at the end of each epoch.

Additionally, we structure the frontend hash table's overflow list similar to prior nonblocking concurrent lists [17] so that a reader does not get stuck in a cycle if the node it is accessing is removed from the list by a concurrent writer. While reads are nonblocking, concurrent writers do synchronize with each other on the bucket's spinlock.

4.3 Managing Writer and Gleaner Counts

In the base design, Bullet contains a static mapping between frontend writers, logs, and backend gleaners. Although this approach avoids synchronization among writers and gleaners, it wastes CPU cycles if there is a mismatch in the rates of log record production and consumption. We need to decouple these three parts of Bullet to let threads dynamically perform the roles of frontend and backend based on the write load.

4.3.1 Decoupling Writers from Gleaners

Maximizing Bullet's throughput requires that we keep all threads busy. In practice, this requires that we relax the 1:1 mapping between writers and gleaners. We permit each writer/gleaner to append/consume entries to/from any log. This way we achieve optimal throughput by setting the writer/gleaner ratio according to the ratio of the respective rates of production/consumption of log entries.

Although this requires synchronization among both writers and gleaners, we make the overhead negligible, by coarsening switching intervals between writers and gleaners. Writers lock their log and keep the lock as long as the log is not full. When a log fills, the writer unlocks it and switches to the next free log not currently in use. The same thing happens for gleaners; they switch logs when they have no work to do. For log sizes on the order of megabytes, these switching events are rare enough not to impact performance in an observable way.

4.3.2 Dynamic Adjustment of Writer/Gleaner Ratio

One drawback of the preceding approach is that, selecting the correct writer and gleaner counts, requires knowing the rates of producing and consuming log entries. However, these rates depend heavily on the workload (read/write ratio, key distribution), and the relative performance of DRAM and persistent memory. For example, a write-heavy workload on a machine with a slow persistent memory generally requires more gleaners than a read-heavy workload.

To achieve high throughput in as many scenarios as possible, threads dynamically change their roles, writing or gleaning depending on what is currently needed. The advantage of this approach is twofold. First, it makes Bullet suitable for a wide range of workloads, without prior profiling and configuration. Second, the system adapts to dynamically changing workload, maintaining near optimal throughput throughout.

The key for achieving optimal throughput is preventing the logs from becoming full (writers stalling) or empty (gleaners stalling). To this end, we periodically check (once per epoch) the occupancy of the logs. If the log occupancy passes a pre-defined threshold of 60%, we switch one thread from writing to gleaning. If, upon the next check, the occupancy is still increasing, we add yet another gleaner. We repeat this until the log occupancy starts decreasing. The inverse happens when the log occupancy drops below 30%, in which case we start moving gleaners back to writing.

Making threads switch between worker and gleaner roles is an interesting control theory problem by itself. Our algorithm evolved over several attempts at simpler approaches, which failed to achieve both stability (i.e., avoid frequent role switching) and responsiveness.

4.4 Collapsing `Put` Operations

Recall that multiple updates to the same key result in a linked list of log records. Gleaners traverse the chain and apply all the log records from oldest to newest (see subsection 2.3).

However, it is not necessary to apply every `Put` operation, since the most recent `Put` overwrites the effects of all older `Puts` and `Removes`; same is the case with `Removes`. Thus, a gleaner applies only the newest operation in a chain of log records, without following back pointers at all. To prevent a newer value being overwritten by an older one, a gleaner applies a log record only if it contains the globally newest update for the corresponding key. To determine whether a log entry is the newest for its key, the gleaner checks the corresponding K-V pair's `lentry` pointer, as this always points to the key's newest log record.

Collapsing updates appears to make the criss-cross log record links unnecessary. However, this is the case only for *idempotent* updates, e.g. `Put` and `Remove`. We however plan to extend Bullet to support non-idempotent updates similar to recent data structure stores like Redis [44], where the criss-cross links will be required for correctness.

5 Recovery and Warmup

Recovery is simple for Bullet. Since updates must complete in the frontend before we apply them to the backend and the frontend disappears on failure, Bullet never has anything to undo. In theory, recovery entails two parts: 1) reinitializing the frontend DRAM resident state and 2) applying log records in the CRLs to the backend. Bullet's architecture however permits us to eliminate all of step 2 from recovery, and reduce step 1 drastically: During recovery, the CRLs' log records can be applied to the frontend hash table, instead of applying them to the backend. This has the nice side effect that there is no special recovery code for the backend. We assume that recovery for the backend's persistent transactions happens before Bullet's recovery is triggered. Application of CRLs to the backend is relegated to the normal gleaning process.

Note that recovery itself "warms up" the frontend hash table with key-value pairs found in the CRLs. Thereafter, misses in the frontend populate the corresponding key-value pairs from the backend as described in subsection 2.3. Thus warmup time and recovery time are one and the same and are proportional to the time taken to apply the CRLs.

6 Implementation Notes

We implemented Bullet in C++ and used our PM access library (section 3) developed in C. We used `pthreads` to implement both the frontend and backend threads. The frontend K-V store uses the `jemalloc` library to handle memory allocations. For the backend, we rely on the access library's heap manager, which is based on the scalable Hoard allocator [4, 50].

The PM access library presents to Bullet a persistent memory hosted `mmap()`ped file as a persistent region. Bullet's persistent domain is precisely that region. The `mmap` dependency means that the address of the persistent domain is unpredictable. Therefore, we must represent persistent pointers in a manner amenable to relocation, so we represent persistent pointers as offsets from the region's base address.

Bullet's backend contains a *root structure* that hosts persistent pointers to the persistent hash table and the cross-referencing logs. Wherever we do not use persistent transactions, we carefully order stores and persists to persistent data structures (e.g. CRL appends, initializing a newly allocated key-value pair) for crash consistency.

All update operations in Bullet's backend threads use transactions to apply CRL log records to the backend hash table. In contrast, Bullet's frontend updates need not be transactional; they need only append records to the the CRLs. This indicates two different implementations for all update operations (e.g., frontend and backend implementations of `Put`, `Remove`, etc. operations). This doubles the coding effort for these operations.

The access library's transactional runtime uses Intel's persistence enforcement instructions [24] – cache-line writeback (`clwb`) and persist barrier (`sfence`) instructions to correctly order transactional writes to PM. CRL appends also use these instructions: first, we write back the cache lines of the updated log record using `clwb` and then persist them using `sfence`. Next, we update and persist the log's tail index using the same instructions.

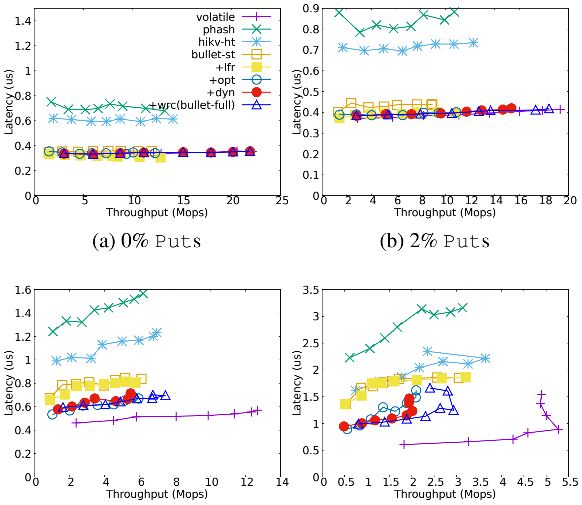
7 Evaluation

We evaluated Bullet's performance on Intel's Software Emulation Platform [43, 57]. This emulator hosts a dual socket 16-core processor, with 512GB of DRAM, of which 384GB is configured as "persistent memory". Persistent memory is accessible to applications via `mmap`ped files hosted in the emulator's PMFS instance [43].

The aforementioned persistence instructions, `clwb` and `sfence`, are not supported by the emulator. We simulated `clwb` with a `nop` and the `sfence` with an idle spin loop of 100 nanoseconds. We expect these to be reasonable approximations since `clwb` is an asynchronous cache line writeback, and an `sfence` ensures that prior writebacks make it to the memory controller buffers, which we assume to be a part of the memory hierarchy's "persistence domain" [45] – 100 nanoseconds is the approximate latency to the memory controller buffers on the emulator. The emulator does support configurable `load` latency to persistent memory; we set it to 300 nanoseconds, twice the load latency of the DRAM on the machine [57]. We configured the PM to have the same bandwidth as that of the emulator's DRAM. We experimented with a lower bandwidth option (1/4 of DRAM bandwidth, which was the only other available option on the emulator), but obtained identical results, suggesting that our experiments did not saturate the memory bandwidth available on the emulator (36 GB/s).

We conducted an 8-way evaluation to see how effectively Bullet eliminates the gap between DRAM and PM performance. The eight systems were as follows. 1) A DRAM-only version that uses just the frontend hash table (volatile), which places an upper bound on performance. 2) A PM-only version that uses Bullet's backend hash table (phash), providing a lower bound on performance. 3) hikv-ht, our implementation of the hash table component of HiKV – a state-of-the-art K-V store, whose hash table resides in PM [54]. HiKV gets a somewhat unfair advantage in our experiments, because it does not ensure that the state of the persistent memory allocator persists. However, the allocator's state can be rebuilt after a restart from HiKV's hash table, although we have not implemented this. 4) bullet-st, the base version of Bullet, which assigns frontend and backend threads statically and uses transactions in the critical path of update operations. 5) +lfr, the base version of Bullet with optimized, lock-free `Gets`. 6) +opt, the version of Bullet that additionally eliminates failure atomic transactions from the critical path of update requests. 7) +dyn, the Bullet version that, along with above optimizations, supports dynamic thread switching between the frontend and backend. 8) bullet-full (also appears as +wrc(bullet-full) in the graphs), the full Bullet version that additionally contains the write collapsing optimization. Although the frontend of Bullet can be a subset of the backend, in our experiments the frontend is a full copy of the backend.

We evaluate various aspects of Bullet comprising scalability and latency, dynamic behavior of worker threads, and log size sensitivity in a microbenchmark setting. In all our experiments, `Get`/`Put` requests are drawn from a pre-created stream of inputs with a zipfian distribution of skewness 0.99, which is the same as YCSB's input distribution [11]. We average over five test runs for each data point. We also use an evaluation framework that uses independent clients to better understand end-to-end performance of these systems as client load increases. The clients are independent threads residing in the same address space as Bullet and communicate requests and responses through globally shared request/response buffers. We do not use clients communicating with Bullet over TCP connections, since the network stack latency itself tends to significantly mute important performance trade offs between the evaluated K-V stores [14, 54].



(a) 0% `Puts`

(b) 2% `Puts`

(c) 15% `Puts`

(d) 50% `Puts`

Figure 5: Latency (99^{th} percentile) vs. Throughput results. Each point on the curves represents a different number of worker threads ranging from 2 to 16 in increments of 2.

7.1 Latency vs. Throughput

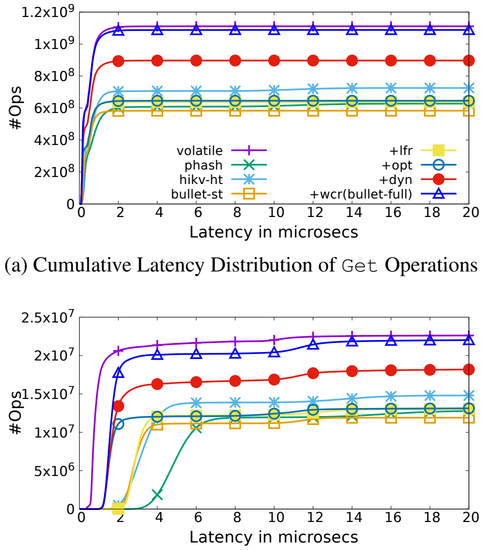
Figure 5 shows performance as a latency/throughput tradeoff under workloads whose write percentage varies from 0% (read-only) to 50% (write-heavy). We begin by creating a 50-million key/value pair store with 16-byte keys and 100-byte data values; these choices are in line with what is observed in real-world settings [2, 54]. Each experiment runs a specified number of of worker threads with the requested read/write ratio, using `Get`/`Put` operations (`Remove` performance is comparable to that of `Put`). Each worker selects key-value pairs from the prepopulated zipfian stream of keys and performs the selected operation. The worker continuously repeats these operations for 1 minute (we experimented with 5 – 10 minute runs, but the results were unchanged).

For the dynamic worker role versions of Bullet (+dyn and bullet-full), some workers switch roles to become backend log gleaners. In such cases, the worker posts its current unapplied operation on a globally visible queue of requests, so that some other frontend worker will process it (to ensure forward progress, we guarantee that at least 1 worker remains in the frontend). We measure latency of only those operations that have a frontend worker assigned to them (the requests posted in the central queue are a rare occurrence and are processed relatively immediately by frontend worker threads).

Notice the clear impact of slower PM on the 0% `Put` case in Figure 5a. The difference between phash's and volatile's latency and throughput mirrors the difference in PM and DRAM latency. hikv-ht performs noticeably better than phash, owing to some of its cache locality oriented optimizations. But these marginal improvements suggest that additional optimizations cannot eliminate the fundamental problem of slower PM. All of Bullet's versions' latencies align almost exactly with volatile. bullet-st shows slight overhead associated with lock-based Gets. All static worker role assignment variants of Bullet (bullet-st, +lfr, +opt) effectively end up using just half of the available workers in the frontend and produce throughput approximating half the throughput of volatile; the backend worker threads effectively waste CPU cycles. Our dynamic worker assignment framework (in +dyn, +wrc(bullet-full)) correctly assigns all workers to the frontend, which performs comparably to volatile.

The 2% Put test is more representative of real-world (read-dominated) workloads [40]. As Figure 5b shows, the relative latency differences remain similar; there is a small increase in the absolute latencies reflecting effects of longer latency Put operations. For the same reason, the absolute throughput numbers are smaller, but the relative difference between volatile, phash, hikv-ht, and the static variants of Bullet remains the same. However, in +dyn and bullet-full we begin to see the impact of logging. The primary source of these overheads is the dynamic switching of 1 or 2 worker threads between the frontend and backend. Note that even with 2% Puts, our CRLs quickly cross the occupancy threshold of 60%, which forces frontend threads to incrementally switch to the backend log gleaner roles if the occupancy keeps growing across epochs. A consistent rate of 2% Put traffic is large enough to force at least one worker to stay a log gleaner through the entire execution. +dyn's performance drops by a significant 25% compared to volatile. However, our write collapsing optimization works exceptionally well to significantly reduce that margin to about 5%: the zipfian distribution of requests allows for substantial write collapsing (30 – 50%), which leads to the log gleaner applying the log more quickly, spending the saved time in frontend request processing.

The 15% workload, shown in Figure 5c, illustrates more clearly the impact of the different optimizations. Compared to volatile, Bullet's bullet-st and +lfr versions show a 40% degradation in latency. The failure atomic transactions used for Put operations of these versions are primarily responsible for this degradation. This degradation is mitigated by half with our critical path optimization present in Bullet's +opt, +dyn, and bullet-full versions. Latency of the PM-only K-V stores, phash and hikv-ht, is approximately $3X$ and $2.5X$ higher than that of volatile. Notice the throughput of Bullet's dynamic versions drops significantly. With 15% Puts, we observed



(a) Cumulative Latency Distribution of Get Operations

(b) Cumulative Latency Distribution of Put Operations

Figure 6: Get, Put Cumulative Latency distributions on 16-thread test runs with 2% Puts.X

a larger fraction (4 – 6) of worker threads getting forced to operate as log gleaners in the backend for the entire duration of the test. That leads to a significant reduction in overall throughput, since threads migrated from the frontend to the backend do not process new requests.

With the even higher 50% Put rate of Figure 5d, we observe additional interesting behavior. The variants that use transactions in their Put critical paths exhibit significantly increased latency, approaching that of hikv-ht's latency. The rest of Bullet's variants (+opt, +dyn, and bullet-full) exhibit lower latency, which starts to grow only as the set of worker threads grows. We attribute this performance degradation to cache contention between frontend and backend threads. Notice that the working sets of the frontend and backend threads are largely different – a frontend log writer accesses the frontend hash table and a log, whereas the colocated (on the same socket) backend log gleaner accesses the backend hash table and possibly a different log. The more threads there are, the greater the cache contention, and the worse the performance. Overall, the results suggest that workloads with very high write rates are not a good fit for Bullet.

7.2 Latency Distribution of Gets and Puts

Figure 6's segregated cumulative latency distribution graphs for Gets and Puts provide deeper insight into the behaviour of the K-V stores. Figure 6a shows latency of Gets. The phash and hikv-ht latencies average to about 450 and 380 nanoseconds respectively, whereas volatile and all of Bullet's versions average to 220 nanoseconds. Average latencies of Puts are more scattered: volatile is the fastest with 750 nanoseconds, followed by Bullet's versions that do not contain trans-

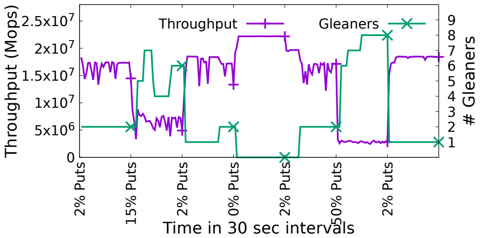


Figure 7: Variation in throughput and log gleaner count as the write load on bullet-full changes. The load, shown on the X-axis labels, switches every 30 seconds. The left Y-axis shows throughput for every second, and the right Y-axis shows the number of log gleaners at the end of each second.

actions in the critical path (at 1 microsecond), followed by Bullet's versions that contain transactions in the critical path (at 2.5 miroseconds). HiKV's latency matches that of Bullet's versions with transactions on the critical path. phash is the slowest with latency averaging to 5 microseconds; this is an $8X$ slowdown compared to volatile. Note that the backend `Put` operations in all of Bullet's versions apply the same `Put` operation used in phash. This largely explains the significantly higher cost associated with applying log records to the backend, and why as little as 2% `Puts` can force worker threads to play the log gleaner role for much longer durations that amplify to a minimum of 10% slowdown in throughput compared to volatile in Figure 5b.

7.3 Dynamic Behavior of Workers

Figure 7 shows bullet-full's dynamic worker role framework in action. It reports the throughput as well as the gleaner count at the end of every second, over a duration of 210 seconds. Every 30 seconds, we change the load of `Puts` on bullet-full. After a warmup phase of 30 seconds of 2% `Put` rate, we vary the `Put` rate between 2-15-2-0-2-50-2%, in that order. As is clear from the graph, our dynamic worker role adaptation strategy works well in adapting to the changing load of `Puts`. At times, as observed in the 15% and 50% `Put` phases, our adaptation algorithm fluctuates around the optimal mix of frontend and backend workers before converging to a stable mix that matches frontend producers of log records with backend gleaners that consume these log records.

Throughout the execution, for 2% `Puts`, the throughput hovers around 16 Mops, and the number of log gleaners ranges from $1-2$. This helps explain the reduction in observed throughput of bullet-full compared to the throughput of volatile in Figure 5b. After a switch to a 15% `Put` rate, the throughput switches immediately, reflecting the corresponding uptick in the gleaner count. For the 0% `Put` case, our algorithm quickly and correctly converges to a gleaner count of 0, thus explaining the throughput reported in Figure 5a that matches the throughput of volatile. For the 15% and 50% `Put` cases, the number of gleaners needed settles down to 6 and 8 respectively. Note that in our 100% `Put` experiments (not reported here in detail), we observed the number of gleaners vary between $13-15$.

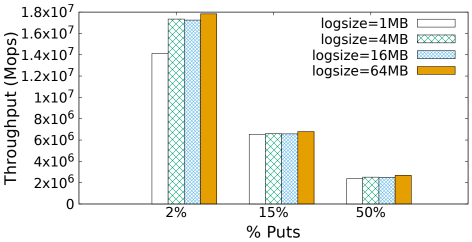


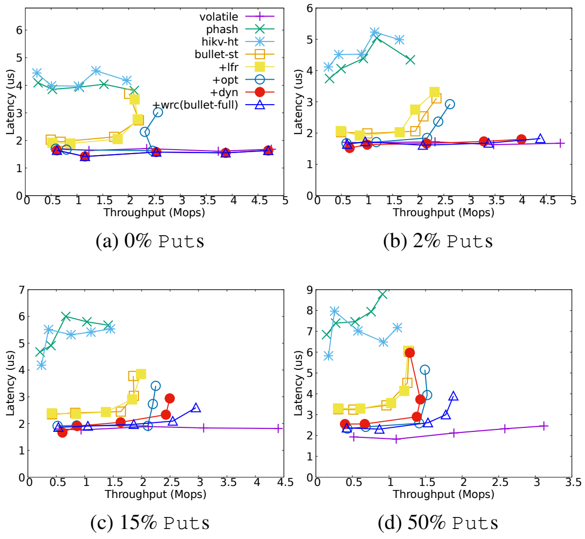
Figure 8: Effect of log size (per thread) on throughput of bullet-full.

7.4 Log Size Sensitivity

Bullet's CRLs act as speed matching buffers between the frontend and backend worker threads. As long as there is enough available space in CRLs, frontend workers keep appending log records as quickly as they can. When CRL occupancy gets too high, workers are incrementally switched to the backend to match the frontend load of CRL population. If the CRLs are too small in size, Bullet can easily enter a mode where threads bounce between frontend and backend at a high frequency, which in turn could lead to significant disruption in overall performance. The question then to consider is – how big should these logs be to avoid performance degradation due to workers switching frontend and backend roles?

To that end, Figure 8 shows the results of our experiment where we vary the per-thread log size from 64 MBs (the size we used for all experiments described above), down to 1 MB. In addition, the CRL infrastructure maintains 32 logs in-all; when a frontend worker exhausts its log, it can switch to another log that is not in use by another frontend worker. As a result, per-thread log sizes of 1, 4, 16, and 64 MBs result in total CRL footprint of 32, 128, 512, and 2048 MBs respectively. Even the largest 2048 MB CRL footprint may be acceptable in a future PM-equipped system that hosts multi-terabytes of PM.

The overall results were quite surprising to us: We expected log size to have a big impact on performance across the board. However, for write-intensive workloads, the log size does not matter to throughput. The `Put` load is high enough that the system converges to a stable mix of frontend and backend threads. The interesting case is 2% `Puts`. We observe a modest 3% drop in throughput when we transition from 64 MB logs to 4 or 16 MB logs, whereas a further reduction in log size (to 1 MB) results in a significant 20% drop in throughput. The problem with 1 MB logs is that the `Put` load generates enough log traffic to populate CRLs quickly enough that worker threads switch to the backend more aggressively than is necessary. Subsequently, a high number of of backend workers drains the log quickly after which a larger than necessary fraction of backend workers switch



(a) 0% Puts

(b) 2% Puts

(c) 15% Puts

(d) 50% Puts

Figure 9: Latency (99^{th} percentile) vs. Throughput results for test runs with independent client threads (1,2,4,6,8). The number of workers is kept to a constant 8. The graphs show the effect of increasing client load on Bullet.

to the frontend role. This over-aggressive switching of worker roles results in the performance degradation. However, 4 MB log size is big enough to absorb the log population rate more gracefully. Note that the size of each log record (including its header) is 193 bytes.

7.5 End-to-End Performance

To understand the end-to-end performance observed by independent clients, we conducted an experiment where clients generated back-to-back requests based on the zipfian distribution mentioned earlier. The clients were hosted as independent threads in Bullet's address space, eliminating the overheads related to network latencies.

Each client generates a request in its local buffer that is visible to all of Bullet's workers (but not other clients), waits for a response from Bullet, and repeats. The workers synchronize amongst each other, using a per buffer lock, to get and process client requests. We reduce contention on these locks by ensuring that workers serve a multitude of requests (1,000 in our experiments) before releasing an acquired lock and switching over to another buffer. To minimize interference between workers and client threads, we host the workers on one socket of the emulator and the client threads reside on the other socket. We effectively end up getting a maximum of 8 worker threads for each test run in this experiment.

Figure 9 shows performance of the various K-V stores with growing number of client threads. First, notice the $5X$ increase in latency of operations over all the K-V stores compared to earlier experiments (Figure 5). This slowdown was a big surprise. However, additional experimentation revealed cross-socket cache access latencies to be the biggest contributor to the overheads: when we pinned communicating workers and client threads on the same socket the latency increase reduced to approximately 10%. We did not pursue such an intermingled topological layout for clients and workers since workers tend to dynamically switch between clients when some workers are busy performing gleaning operations, which led to unpredictable performance.

Other than the unexpected NUMA effects on performance, the observed relative degradation in latencies of Bullet's flavors bullet-st, +lfr, and +opt appears to be much greater than our prior experiments (Figure 5). This degradation can be squarely attributed to the fact that these flavors of Bullet are effectively left with 4 frontend workers, and a greater number of clients (up to 8) results in overload leading to higher latencies at client counts greater than 4. Similar relative latency degradation can be observed in the 15% and 50% write loads for Bullet flavors +dyn and +wrc(bullet-full) : Some worker threads are forced to play the backend gleaner role, which increases the load on the frontend workers since the number of clients is now greater than the frontend workers.

In general, since writes are expensive, an increasing percentage of writes tends to reduce the performance gains we get from the two-tiered architecture of Bullet. We conclude that Bullet does not really close the performance gap between volatile and persistent K-V stores for write-heavy workloads. However, it significantly closes this performance gap in read-dominated workloads.

8 Conclusion

While emerging byte-addressable persistent memory technologies, such as Intel/Micron's 3D XPoint, will approach the performace of DRAM, we expect to see a non-trivial performance gap (within an order of magnitude) between them. We showed that this performance gap can have significant implications on the performance of persistent memory optimized K-V stores. In particular, we conclude that DRAM does have a critical performance role to play in the new world dominated by persistent memory. We presented our new K-V store, called Bullet, that is architected to exploit this exact observation.

We introduced *cross-referencing logs (CRLs)*, a general purpose scalable logging framework that can be used to build a two-tiered architecture for a persistent K-V store that leverages capabilities of emerging byte-addressable persistent memory technologies, and the much faster DRAM, to deliver performance approaching that of a DRAM-only K-V store for read-dominated workloads. Our performance evaluation shows the effectiveness of Bullet's architectural features that bring its performance close to that of a DRAM-only K-V store for read-heavy workloads. Write-heavy workloads' performance is severely limited by the high latency of *failure-atomic* writes, and further research is warranted to reduce these overheads.

References

[1] 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[2] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[3] Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of 1st Workshop on Interactions of NVM-Flash with Operating Systems and Workloads* (2013).

[4] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 117–128.

[5] Blott, M., Karras, K., Liu, L., Vissers, K. A., Bär, J., and István, Z. Achieving 10gbps line-rate key-value stores with fpgas. In *5th USENIX Workshop on Hot Topics in Cloud Computing* (2013).

[6] Bridge, B. Nvm-direct library. *https://github.com/oracle/nvm-direct*, 2015.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

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

[9] Chi, P., Lee, W., and Xie, Y. Making b$^+$-tree efficient in pcm-based main memory. In *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014* (2014), pp. 69–74.

[10] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 105–118.

[11] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), pp. 143–154.

[12] Debnath, B., Haghdoost, A., Kadav, A., Khatib, M. G., and Ungureanu, C. Revisiting hash table design for phase change memory. *SIGOPS Oper. Syst. Rev. 49*, 2 (Jan. 2016), 18–26.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007* (2007), pp. 205–220.

[14] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 401–414.

[15] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 371–384.

[16] Giles, E., Doshi, K., and Varman, P. J. Softwrap: A lightweight framework for transactional support of storage class memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–14.

[17] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings* (2001), pp. 300–314.

[18] Apache HBase. *http://hbase.apache.org/.*

[19] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing* (2003), pp. 92–101.

[20] Hetherington, T. H., O'Connor, M., and Aamodt, T. M. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), pp. 43–57.

[21] Hosomi, M., Yamagishi, H., Yamamoto, T., Bessho, K., Higo, Y., Yamane, K., Yamada, H., Shoji, M., Hachino, H., Fukumoto, C., Nagao, H., and Kano, H. A novel non-volatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting* (2005), 459–462.

[22] Hu, W., Li, G., Ni, J., Sun, D., and Tan, K.-L. BP-Tree : A Predictive B+-Tree for Reducing Writes on Phase Change Memory. *IEEE Transactions on Knowledge and Data Engineering 26* (2014), 2368–2381.

[23] Huai, Y. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin 18*, 6 (2008), 33–40.

[24] Intel® 64 and IA-32 Architectures Software Developer's Manual. *http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf*, 2015.

[25] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Aether: A Scalable Approach to Logging. *Proceedings of VLDB Endowment 3*, 1-2 (Sept. 2010), 681–692.

[26] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., and Ailamaki, A. Scalability of Write-ahead Logging on Multicore and Multisocket Hardware. *The VLDB Journal 21*, 2 (Apr. 2012), 239–263.

[27] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 295–306.

[28] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[29] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[30] LevelDB – a fast and lightweight key/value database library. *http://leveldb.org/.*

[31] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. Architecting to achieve a billion requests per second

throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (2015), pp. 476–488.

[32] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), pp. 429–444.

[33] LIM, K., MEISNER, D., SAIDI, A. G., RANGANATHAN, P., AND WENISCH, T. F. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), pp. 36–47.

[34] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), pp. 183–196.

[35] MARATHE, V. J., MISHRA, A., TRIVEDI, A., HUANG, Y., ZAGHLOUL, F., KASHYAP, S., SELTZER, M., HARRIS, T., BYAN, S., BRIDGE, B., AND DICE, D. Persistent Memory Transactions *https://arxiv.org/abs/1804.00701*, 2018.

[36] Memcached – a distributed memory object caching system. *https://memcached.org/*.

[37] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), pp. 103–114.

[38] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (1992), 94–162.

[39] NAWAB, F., IZRAELEVITZ, J., KELLY, T., MORREY, C. B., CHAKRABARTI, D., AND SCOTT, M. L. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Synposium on Distributed Computing* (2017).

[40] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 385–398.

[41] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. 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* (2016), pp. 371–386.

[42] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014* (2014), pp. 265–276.

[43] RAO, D. S., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014* (2014), p. 15.

[44] Redis – in-memory data structure store, *http://redis.io/*.

[45] RUDOFF, A. Deprecating the PCOMMIT Instruction. *https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction*, 2016.

[46] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing Memristor found. *Nature 453* (2008), 80–83.

[47] SUZUKI, K., AND SWANSON, S. A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014. In *2015 IEEE International Memory Workshop* (2015), pp. 1–4.

[48] Swift Object Store. *https://swift.openstack.org/*.

[49] THE SNIA NVM PROGRAMMING TECHNICAL WORKING GROUP. NVM Programming Model (Version 1.0.0 Revision 10), Working Draft. http://snia.org/sites/default/files/NVMProgrammingModel_v1r10DRAFT.pdf, 2013.

[50] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), pp. 91–104.

[51] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), pp. 16:1–16:14.

[52] WANG, T., AND JOHNSON, R. Scalable logging through emerging non-volatile memory. *PVLDB 7*, 10 (2014), 865–876.

[53] WU, X., ZHANG, L., WANG, Y., REN, Y., HACK, M., AND JIANG, S. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 14:1–14:15.

[54] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 349–362.

[55] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), pp. 167–181.

[56] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[57] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015* (2015), pp. 1–10.

[58] ZHOU, J., SHEN, Y., LI, S., AND HUANG, L. NVHT: An Efficient Key-value Storage Library for Non-volatile Memory. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies* (2016), pp. 227–236.