Reaping the performance of fast NVM storage with uDepot

Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas[*]
IBM Research, Zurich
{*kou, nio, iko*}*@zurich.ibm.com*

Abstract

Many applications require low-latency key-value storage, a requirement that is typically satisfied using key-value stores backed by DRAM. Recently, however, storage devices built on novel NVM technologies offer unprecedented performance compared to conventional SSDs. A key-value store that could deliver the performance of these devices would offer many opportunities to accelerate applications and reduce costs. Nevertheless, existing key-value stores, built for slower SSDs or HDDs, cannot fully exploit such devices.

In this paper, we present uDepot, a key-value store built bottom-up to deliver the performance of fast NVM block-based devices. uDepot is carefully crafted to avoid inefficiencies, uses a two-level indexing structure that dynamically adjusts its DRAM footprint to match the inserted items, and employs a novel task-based IO run-time system to maximize performance, enabling applications to use fast NVM devices at their full potential. As an embedded store, uDepot's performance nearly matches the raw performance of fast NVM devices both in terms of throughput and latency, while being scalable across multiple devices and cores. As a server, uDepot significantly outperforms state-of-the-art stores that target SSDs under the YCSB benchmark. Finally, using a Memcache service on top of uDepot we demonstrate that data services built on NVM storage devices can offer equivalent performance to their DRAM-based counterparts at a much lower cost. Indeed, using uDepot we have built a cloud Memcache service that is currently available as an experimental offering in the public cloud.

1 Introduction

Advancements in non-volatile memory (NVM) technologies enable a new class of block-based storage devices with unprecedented performance. These devices, which we refer to as Fast NVMe Devices (FNDs), achieve hundreds of thousands of IO operations per second (IOPS) as well as low latency, and constitute a discrete point in the performance/cost tradeoff spectrum between DRAM and conventional SSDs. To illustrate the difference, the latency of fetching a 4 KiB block in conventional NVMe Flash SSD is 80 µs, while in FNDs the same operation takes 7 µs (Optane drive [88]) or 12 µs (Z-SSD [48,74]). To put this in perspective, a common round-trip latency of a TCP packet over 10 Gigabit Ethernet is 25 µs-50 µs, which means that using FNDs in commodity datacenters results in storage no longer being the bottleneck.

Hence, FNDs act as a counterweight to the prevalent architectural trend of data stores placing all data in main memory [26,35,72,73,78]. Specifically, many key-value (KV) stores place all their data in DRAM [21,25,44,52,57,59,68,73] to meet application performance requirements. An FND-based KV store offers an attractive alternative to DRAM-based systems in terms of cost and capacity scalability.[1] We expect that many applications, for which conventional SSDs are not performant enough, can now satisfy their performance requirements using KV stores built on FNDs. In fact, since for many common setups FNDs shift the bottleneck from storage to the network, it is possible for FND-based KV stores to provide equivalent performance to that of their DRAM-based counterparts.

Existing KV stores, however, cannot use FNDs to their full potential. First, KV stores that place all their data in DRAM require OS paging to transparently use FNDs, which results in poor performance [33]. Second, KV stores that place their data in storage devices [8,24,31,50], even those that specifically target conventional SSDs [3,19,20,58,60,84,87,91], are designed with different requirements in mind: slower devices, smaller capacity, and/or no need to scale over multiple devices and cores. As Barroso et al. [7] point out, most existing systems under-perform in the face of IO operations that take a few microseconds.

Motivated by the above, we present uDepot, a KV store designed from the ground up to deliver the performance of FNDs. The core of uDepot is an embedded store that can

[*]Now at Google.

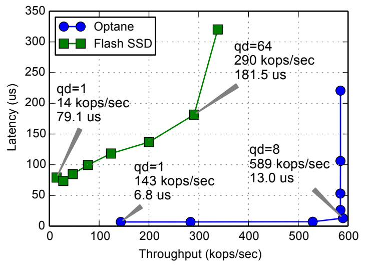
[1]At the time of writing: DRAM costs about $10/GiB, an Optane NVMe drive $1.25/GiB, and a commodity Flash NVMe drive $0.4/GiB.

be used by applications as a library. Using this embedded store we build two network services: a distributed KV store using a custom network protocol, and a distributed cache that implements the Memcache [64] protocol, which can be used as a drop-in replacement for memcached [65], a widely used [5, 70] DRAM-based cache.

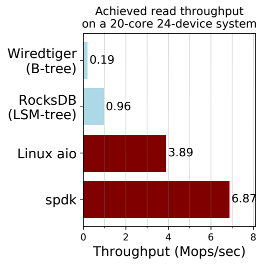
By design, uDepot is *lean*: it provides streamlined functions for efficient data access, optimizing for performance instead of richer functionality (e.g., range queries). uDepot is *efficient* in that it: i) achieves low latency, ii) provides high throughput per core, iii) scales its performance with the number of drives and cores, iv) enforces low bounds to end–to-end IO amplification, in terms of bytes and number of operations, and, finally, v) achieves a high utilization of storage capacity. This requires multiple optimizations throughout the system, but two aspects are especially important. First, efficiently accessing FNDs. Most existing KV stores use synchronous IO that severely degrades performance because it relies on kernel-scheduled threads to handle concurrency. Instead, uDepot employs asynchronous IO and, if possible, directly accesses the storage device from user-space. To this end, uDepot is built on TRT, a task runtime for IO at the microsecond scale that uses user-space collaborative scheduling. Second, uDepot uses a high-performance DRAM index structure that is able to match the performance of FNDs while keeping its memory footprint small. (A small memory footprint leads to efficient capacity utilization because less DRAM is needed to index the same storage capacity.) uDepot's index structure is resizable, adapting its memory consumption to the number of items stored. Resizing does not require any IO operations, and is performed incrementally so that it causes minimal disruption.

In summary, our contributions are: **1)** uDepot, a KV store that delivers the performance of FNDs, offering low latency, high throughput, scalability, and efficient use of CPUs, memory, and storage. **2)** TRT, a task run-time system suitable for IO at the microsecond scale, which acts as a substrate for uDepot. TRT provides a programmer-friendly framework for writing applications that fully exploit fast storage. **3)** uDepot's index data structure that enables it to meet its performance goals while being space efficient, dynamically resizing to match the number of KV pairs stored. **4)** An experimental evaluation demonstrating that uDepot matches the performance of FNDs, which, to our knowledge, no existing system can. Indeed, uDepot vastly outperforms SSD-optimized stores by up to $\times 14.7$ but also matches the performance of a DRAM-backed memcached server allowing it to be used as a Memcache replacement to dramatically reduce cost. A cloud Memcache service built using uDepot is available as an experimental offering in the public cloud [39].

The rest of the paper is organized as follows. We motivate our work in §2, discuss TRT in §3, and present and evaluate uDepot in §4 and §5, respectively. In §6 we discuss related work and conclude in §7.



(a) Latency and throughput of 4 KiB random reads on two NVMe devices: a NVMe Flash SSD, and an Optane, as we vary the queue depth (operations in flight) in powers of two using SPDK's perf benchmark [86].



(b) Aggregate throughput of random 4 KiB reads using different IO facilities (aio, spdk) and storage engines (WiredTiger, RocksDB).

Figure 1

2 Background and Motivation

In 2010, arguing for an in-memory KV store, Ousterhout et al. predicted that "Within 5–10 years, assuming continued improvements in DRAM technology, it will be possible to build RAM-Clouds with capacities of 1–10 Petabytes at a cost less than $5/GB" [72]. Since then, researchers have been conducting an "arms race" to maximize performance for in-memory KV stores [10, 21, 44, 52, 57, 68, 73]. In contrast to the above prediction, however, DRAM scaling is approaching physical limits [69] and DRAM is becoming more expensive [22, 40]. Hence, as capacity demands increase, memory KV stores rely on scaling out to achieve the required storage capacity by adding more servers. Naturally, this is inefficient and comes at a high cost as the rest of the node (CPUs, storage) remains underutilized and resources required to support the additional nodes need to increase proportionally as well (space, power supplies, cooling). In addition, while the performance of memory KV stores is impressive, many depend on high-performance or specialized networking (e.g., RDMA, FPGAs) and cannot be deployed in commodity datacenter infrastructures such as the ones offered by many public cloud providers.

Fast NVMe devices (FNDs) that were released recently offer a cost-effective alternative to DRAM, with significantly better performance than conventional SSDs (Fig. 1a). Specifically, the Optane drive, based on 3D XPoint (3DXP),[2] delivers a throughput close to 0.6 Mops/s, and achieves read access latencies of 7 µs, an order of magnitude lower than conventional SSDs, which have latencies of 80 µs or higher [34]. Furthermore, Samsung announced availability of Z-SSD, a new device [89] that utilizes Z-NAND [74] and has similar performance characteristics to Optane, achieving read access latencies of 12 µs. Hence, a KV store effectively using FNDs offers an attractive alternative to its

[2] 3DXP is also used to build devices accessible as memory that offer even lower latencies. Our work focuses on IO devices because they are widely available and the most cost effective option.

DRAM counterparts. This is especially true in environments with commodity networking (e.g., 10 Gbit/s Ethernet) where FNDs shift the bottleneck from the storage to the network, and the full performance of DRAM KV stores cannot be obtained over the network.

Existing KV stores are built with slower devices in mind and fail to deliver the performance of FNDs. As a motivating example, we consider a multi-core and multi-device system aimed at minimizing cost with 20 cores and 24 NVMe drives, and compare the performance of the devices against the performance of two ubiquitous storage engines: RocksDB and WiredTiger. These engines epitomize modern KV store designs, using LSM- and B-trees. We measure device performance with microbenchmarks using the Linux asynchronous IO facility (aio) and SPDK, a library for directly accessing devices from user-space. For the two KV stores, we load 50M items of 4 KiB and measure the throughput of random `GET` operations using their accompanying microbenchmarks while setting appropriate cache sizes so that requests are directed to the devices. Even after tuning RocksDB and WiredTiger to the best of our ability, we were not able to exceed 1 Mops/s and 120 Kops/s, respectively. On the other hand, the storage devices themselves can provide 3.89 Mops/s using asynchronous IO and 6.87 Mops/s using user-space IO (SPDK). (More details about this experiment and how uDepot performs in the same setup can be found in §5.)

Overall, these stores underutilize the devices and even though experts can probably tune them to improve their performance, there are fundamental issues with their design. First, these systems, built for slower devices, use synchronous IO which is highly problematic for IO at the microsecond scale [7]. Second, they use LSM- or B-trees which are known to cause significant IO amplification. In the previous experiment, for example, RocksDB IO amplification was $\times 3$ and WiredTiger's $\times 3.5$. Third, they cache data in DRAM which requires additional synchronization but also limits scalability due to memory requirements, and finally they offer many additional features (e.g., transactions) which may have a toll on performance.

uDepot follows a different path: it is built bottom-up to deliver the performance of FNDs (e.g., by eliminating IO amplification), offers only the basic operations of a KV store, does not cache data, and uses asynchronous IO via TRT, which we describe next.

3 TRT: a task run-time system for fast IO

Broadly speaking, there are three ways to access storage: synchronous IO, asynchronous IO, and user-space IO. The majority of existing applications access storage via synchronous systems calls (e.g., `pread`, `pwrite`). As it is already well established for networking [45], synchronous IO does not scale because handling concurrent requests requires one thread for each, leading to context switches that degrade performance when the number of in-flight requests is higher than the number of cores. Hence, as with network programming, utilizing the performance of fast IO devices requires utilizing asynchronous IO [7]. For example, Linux AIO [43], allows multiple IO requests (and their completions) to be issued (and received) in batches from a single thread. Performing asynchronous IO in itself, however, is not enough to fully reap the performance of FNDs. A set of new principles have emerged for building applications that efficiently access fast IO devices. These principles include removing the kernel from the datapath, favouring polling over interrupts, and minimizing, if not precluding, cross-core communication [9, 75]. While the above techniques initially targeted mostly fast networks, they also apply to storage [47, 94]. In contrast to Linux AIO that is a kernel facility, user-space IO frameworks such as SPDK [85], allow maximizing performance by avoiding context switches, data copying, and scheduling overheads. On the other hand, it is not always possible to use them because they require direct (and in many cases unsafe) access to the device and many environments (e.g., cloud VMs) do not (yet) support them.

Hence, an efficient KV store (or a similar application) needs to access both the network and the storage asynchronously, potentially using user-space IO if available to maximize performance. Existing frameworks, such as `libevent` [55], are ill-suited for this use-case because they assume a single endpoint for the application to check for events (e.g., the `epoll_wait` [46] system call). When combining both access to the storage and network, multiple event (and event completion) endpoints that need to be checked might exist. For example, it might be that `epoll_wait` is used for network sockets, and `io_getevents` [42] or SPDK's completion processing call is used for storage. Furthermore, many of these frameworks are based on callbacks which can be troublesome to use due to the so-called "stack ripping" problem [1, 49].

To enable efficient, yet programmer-friendly, access to FNDs, we developed TRT, a Task-based Run-Time system, where tasks are collaboratively scheduled (i.e., no preemption) and each has its own stack. TRT spawns a number of threads (typically one per core) and executes a user-space scheduler on each. The scheduler executes in its own stack. Switching between the scheduler and tasks is lightweight, consisting of saving and restoring a number of registers without involving the kernel. In collaborative scheduling, tasks voluntarily switch to the scheduler via executing proper calls. An example of such a call to the scheduler is `yield` that defers execution to the next task. There are also calls to spawn tasks, and synchronization calls: waiting and notifying. The synchronization interface is based on `Futures` [29, 30]. Because TRT tries to avoid cross-core communication as much as possible, it provides two variants for the synchronization primitives: intra- and inter-core. Intra-core primitives are more efficient because they do not require synchronization to protect against concurrent access as long as critical sections do not include commands that switch to the scheduler.

Based on the above primitives, TRT provides an infrastructure for asynchronous IO. In a typical scenario, each network connection would be served by a different TRT task. To enable different IO backends and facilities, each IO backend implements a poller task that is responsible for polling for events and notifying tasks to handle these events. To avoid cross-core communication, each core runs its own poller instance. As a result, tasks cannot move across core when they have pending IO operations. Poller tasks are scheduled by the scheduler as any other task.

TRT currently supports four backends: Linux AIO, SPDK (single device and RAID-0 multi-device configurations), and Epoll, with backends for RDMA and DPDK in development. Each backend provides a low-level interface that allows tasks to issue requests and wait for results, and, built on top of that, a high-level interface for writing code resembling its synchronous counterpart. For example, a `trt::spdk::read()` call will issue a read command to SPDK device queues, and call the TRT scheduler to suspend task execution until notified by the poller that processes SPDK completions.

To avoid synchronization, pollers of all backends running on different cores use separate endpoints: Linux AIO pollers use different IO contexts, SPDK pollers use different device queues, and Epoll pollers use a different control file-descriptor.

4 uDepot

uDepot supports `GET`, `PUT`, and `DELETE` operations (§4.5) on variable-sized keys and values. The maximum key and value sizes are 64 KiB and 4 GiB, respectively, with no minimum size for either. uDepot directly operates on the device and does its own (log-structured) space management (§4.1), instead of depending on a filesystem. To minimize IO amplification, uDepot uses a two-level hash table in DRAM as an index structure (§4.2) which allows implementing KV operations with a single IO operation (if no hash collision exists), but lacks support for efficient range queries. The index structure can utilize PBs of storage while still remaining memory efficient by adapting its size to the number of KV entries stored at run-time (resizing). Resizing (§4.3) causes minimal disruption because it is incremental and does not incur IO. uDepot does not cache data and is persistent (§4.4): when a `PUT` (or `DELETE`) operation returns, the data are stored in the device (not in OS cache) and will be recovered in case of a crash. uDepot supports multiple IO backends (§4.6), allowing users to maximize performance depending on their setup. uDepot can currently be used in three ways: as an embedded store linked to the application, as a distributed store over the network (§4.7), or as a cache that implements the Memcache protocol [64] (§4.8).

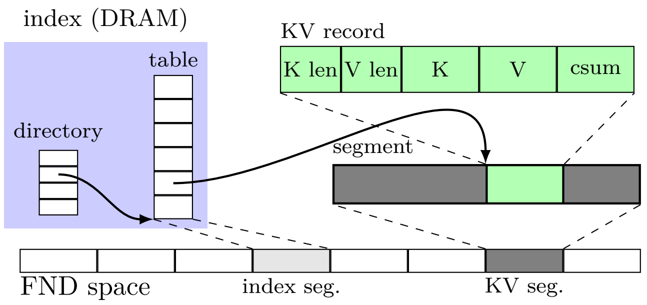


Figure 2: uDepot maintains its index structure (directory and tables) in DRAM. The FND space is split into segments of two types: index segments for flushing index tables, and KV segments for storing KV records.

4.1 Storage device space management

uDepot manages device space using a log-structured approach [67, 79], i.e., space is allocated sequentially and garbage collection (GC) deals with fragmentation. We use this approach for three reasons. First, it achieves good performance on idiosyncratic storage like NAND Flash. Second, it is more efficient than traditional allocation methods even for non-idiosyncratic storage like DRAM [80]. Third, an important use case for uDepot is caching, and there are a number of optimization opportunities when co-designing GC and caches [81, 84]. Allocation is implemented via a user-space port of the log-structured allocator of SALSA [41]. Device space is split into segments (default size: 1 GiB), which are in turn split into grains (typically sized equal to the blocks of the IO device). There are two types of segments: KV segments for storing KV records, and index segments for flushing the index structure to speed up startup (§4.4). uDepot calls SALSA to (sequentially) allocate and release grains. SALSA performs GC and upcalls uDepot to relocate specific grains to free segments [41]. SALSA's GC [76] is a generalized variant of the greedy [12] and circular buffer (CB) [79] algorithms, which augments a greedy policy with the aging factor of the CB.

4.2 Index data structure

uDepot's index is an in-memory two-level mapping directory for mapping keys to record locations in storage (Fig. 2). The directory is implemented as an atomic pointer to a read-only array of pointers to hash tables.

Hash table Each hash table implements a modified hopscotch [37] algorithm, where an entry is stored within a range of consecutive locations, which we call *neighborhood*.[3] Effectively, hopscotch works similarly to linear probe, but bounds probe distance within the neighborhood. If an entry hashes to index `i` in the hash table array, and H is the

[3]The original paper [37] also uses the term "virtual" bucket.

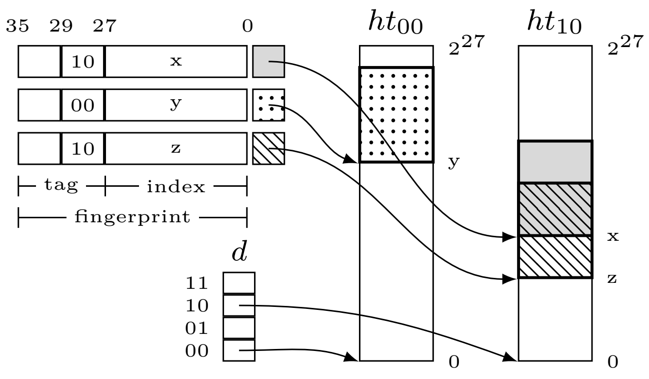


Figure 3: How key fingerprints are used to determine the neighborhood for a key. d is a directory with 4 tables, where only two are shown (ht_{00} and ht_{10}).

neighborhood size (default: 32), then the entry can be stored in any of the H valid entries starting from `i`. In the subsequent paragraphs, we refer to `i` as *neighborhood index*. We choose hopscotch because of its high occupancy, cache efficient accesses, bounded lookup performance – even in high occupancy, and simple concurrency control [21].

We make two modifications to the original algorithm. First, we use a power of two number of entries, indexing the hash table similarly to set-associative caches [38]: we calculate the neighborhood index using the least-significant bits (LSB) of a fingerprint computed from the key. This allows efficiently reconstructing the original fingerprint during resize without needing to fully store it or perform IO to fetch the key and recompute it.

Second, we do not maintain a bitmap per neighborhood, nor a linked-list of entries per neighborhood, that the original algorithm suggests [37]. The latter would increase the memory requirements by 50% for the default configuration (8B entries, and neighborhood size of 32, 4B per entry). A linked list would at least double the memory requirement (assuming 8B pointers and singly or doubly linked list); let alone increase in complexity. Instead of using a bitmap or a list, we perform a linear probe directly on the entries both for lookup and insert.

Synchronization We use an array of locks for concurrency control. These locks protect different regions (*lock regions*) of the hash table, with a region being strictly larger than the neighborhood size (8192 entries by default). A lock is acquired based on the neighborhood's region; if a neighborhood spans two regions, a second lock is acquired in order. (The last neighborhoods do not wrap-around to the beginning of the table so lock order is maintained.) Moreover, to avoid inserts spanning more than two lock regions, we do not displace entries further than two regions apart. Hence, operations take two locks at maximum, and, assuming good key distribution, there is negligible lock contention.

Hash table entry Each hash table entry consists of 8 bytes:

```
struct HashEntry {
  uint64_t neigh_off:5; // neighborhood offset
  uint64_t key_fp_tag:8; // fingerprint MSBs
  uint64_t kv_size:11; // KV size (grains)
  uint64_t pba:40; // storage addr. (grains)
};
```

The `pba` field contains the grain offset on storage where the KV pair resides. To allow utilization of large-capacity devices we use 40 bits for this field, thus able to index petabytes of storage (e.g., 4 PiB for 4 KiB grains). The `pba` value of all 1s indicates an invalid (free) entry.

We use 11 bits to store the size of the KV pair in grains (`kv_size`). This allows issuing a single IO read for `GET`s to KV pairs of up-to 8 MiB when using 4 KiB grains. KV pairs larger than that require a second operation. A valid entry with a KV size of 0 indicates a deleted entry.

The remaining 13 bits are used as follows. The in-memory index operates on a fingerprint of 35 bits, which are the LSBs of a 64 bit cityhash [14] hash of the key (Fig. 3). We divide the fingerprint into a *index* (27 bits) and a *tag* (8 bits). The index is used to index the hash table, allowing for a maximum of 2^{27} entries per table (the default). Reconstructing the fingerprint from a table location requires: i) the offset of the entry within the neighborhood, and ii) the fingerprint tag. We store both on the entry: 8 bits for the tag (`key_fp_tag`), and 5 bits to allow for 32 entries in a neighborhood (`neigh_off`). Hence, if an entry has location λ in the table, then its neighborhood index is $\lambda -$`neigh_off`, and its fingerprint is `key_fp_tag` : ($\lambda -$`neigh_off`).

Capacity utilization Effectively utilizing storage capacity requires being able to address it (`pba` field), but also having enough table entries. Using the LSBs of the tag (8 bits in total) to index the directory, uDepot's index allows for 2^8 tables, each with 2^{27} entries for a total of 2^{35} entries. At the cost of increased collisions, we can further increase the directory by also using up to 5 LSBs from the fingerprint to index it, allowing for 2^{13} tables. We can use up to the 5 neighborhood bits this way because the existing hopscotch collision mechanisms will end up filling positions in the table where no neighborhood starts. If we consider KV pairs with an average size of 1 KiB, this allows utilizing up to 1 PiB ($2^{35+5} \cdot 2^{10}$) of storage. Based on the expected workload and available capacity, users can maximize utilization by configuring the table size parameters accordingly.

Operations For *lookups*, a key fingerprint is generated. We use the fingerprint tag LSBs to index the directory and find the table for this key (if the fingerprint tag is not enough we also use the fingerprint LSBs as described above). Next, we index the table with the fingerprint index to find the neighborhood (also see: Fig. 3). A linear probe is then performed in the neighborhood, and the entries for which the

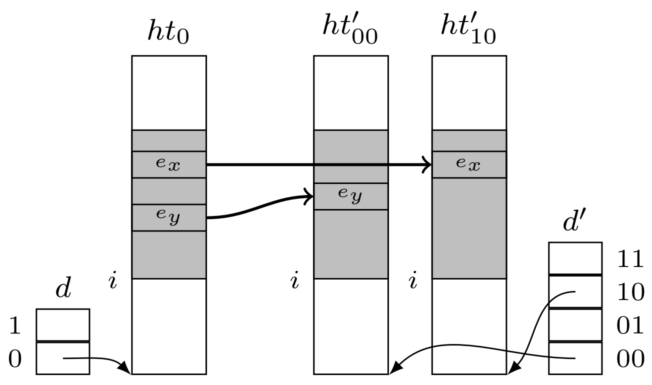


Figure 4: Incremental resizing example, transitioning from a directory with two hash tables (d) to a directory with four (d'). During resizing, insertions copy data from the lock region of ht_0 that contains the neighborhood for the inserted entry, to the same lock regions across two hash tables (ht'_{00}, ht'_{10}).

fingerprint tag (`key_fp_tag`) matches, if any, are returned.

For *inserts*, the hash table and neighborhood are located as described for the lookup. Then a linear probe is performed on the neighborhood and if no existing entry matches the fingerprint tag (`key_fp_tag`), then insert returns the first free entry, if one exists. The user may then fill the entry. If no free entry exists, then the hash table performs a series of displace attempts until a free entry can be found within the neighborhood. If this fails, an error is returned, at which point the caller usually triggers a resize operation. If matching entries exist, then insert returns them. The caller decides whether to update an entry in-place or continue the search for a free entry where they left off.

4.3 Resize operation

The optimal size of the index data structure depends on the number of KV records. Setting the size of the index data structure too low limits the number of records that can be handled. Setting the size too high could waste a significant amount of memory. For example, assuming an average KV record size of 1 KiB, a dataset of 1 PiB would require around 8 TB of memory.

uDepot avoids this issue by dynamically adapting the index data structure to the workload. The resize operation is fast, because it does not require any IO to the device, and causes minimal disruption to normal operations because it is executed incrementally.

The directory grows in powers of two, so that at any point the index holds $n * 2^m$ entries, where m is the number of grow operations, and n is the number of entries in each hash table. We only need the fingerprint to determine the new locations, so no IO operations are required to move hash entries to their new locations. A naive approach would be to move all entries at once, however, it would result in significant delays to user requests. Instead, we use an incremental approach (Fig. 4). During the resize phase, both the new and the old structures are maintained. We migrate entries from the old to the new structure at the granularity of the lock regions. A "migration" bit per lock indicates whether the region has already migrated. An atomic "resize" counter keeps track of whether the total resize operation has concluded, and is initialized to the total number of locks.

Migration is triggered by an insertion operation that fails to find a free entry. The first such failure triggers a resize operation, and sets up a new shadow directory. Subsequent insertion operations migrate all the entries under the locks they hold (one or two) to the new structure, setting the "migration" bit for each lock, and decrementing the "resize" counter (by one or two). Hash tables are pre-allocated during the resize operation in a separate thread to avoid delays. When all entries are migrated from the old to the new structure ("resize" count is zero), the memory of the old structure is released. During the resize operation, lookups need to check either the new or the old structure, depending on the lookup region's "migration" status.

4.4 Metadata and persistence

uDepot maintains metadata at three different levels: per device, per segment, and per KV record. At the device level the uDepot configuration is stored together with a unique seed and a checksum. At each segment's header, its configuration is stored (owning allocator, segment geometry, etc.) together with a timestamp and checksum that matches the device metadata. At the KV record (Fig. 2), uDepot prepends to each KV pair 6B of metadata containing the key size (2B) in bytes, and value size (4B) in bytes, and appends (to avoid the torn page problem) a 2B checksum matching the segment metadata (not computed over the data). The device and segment metadata require 128B and 64B, respectively, are stored in grain aligned locations and their overhead is negligible. The main overhead is due to the per KV metadata which depends on the average key-value size; for a 1 KiB average size the overhead amounts to 0.8%.

To speed up startup, in-memory index tables are flushed to persistent storage, but they are not guaranteed to be up-to-date: the persistent source of truth is the log. Flushing to storage occurs in normal shutdown, but also periodically to speed recovery. Upon initialization, uDepot iterates index segments, restores the index tables, and reconstructs the directory. If uDepot was cleanly shut down (we check this using checksums and unique session identifiers), the index is up to date. Otherwise, uDepot reconstructs the index from KV records found in KV segments. KV records for the same key (new values or tombstones) are disambiguated using segment version information. Because we are not reading data (only keys and metadata) during recovery, starting up after a crash typically takes a few seconds.

4.5 KV operations

For GET, a 64 bit hash of the key is computed and locking of the associated hash table region is performed. A lookup (see §4.2) is performed, returning zero or more matching hash entries. After the lookup, the table's region is unlocked. If no matching entry is found, the key does not exist. Otherwise, the KV record is fetched from storage for each matching entry; either a full key match is found and the value is returned, or the key does exist.

For PUT, we first write a KV record in the log out-of-place. Subsequently, we perform an operation similar to GET (key hash, lock, etc.) to determine whether the key already exists, using the insert (see §4.2) hash table function. If not, we insert a new entry to the hopscotch table if a free entry exists – if no free entry exists, then we trigger a resize operation. If a key already exists, we invalidate the grains of the previous entry, and update the table entry in-place with the new location (pba) and size of the KV record. Note that, also like GET, read IOs to matching hash table entries are performed without holding the table region lock. Unlike GET, though, PUT re-acquires the lock if the record is found, and repeats the lookup to detect concurrent mutation(s) on the same key: if such a concurrent mutation is detected, then the operation that updated the hash table entry first, wins. If the PUT fails, then it invalidates the grains it wrote before the lookup, and returns an appropriate error. PUT updates existing entries by default, but provides an optional argument where the user can choose instead to perform a PUT (i) only if the key exists, or (ii) only if the key does not exist.

DELETE is almost identical to PUT, other than it writes a tombstone entry instead of the KV record. Tombstone entries are used to identify deleted entries on a restore from the log, and are recycled during GC.

4.6 IO backends

uDepot bypasses the page cache and accesses the storage directly (O_DIRECT) by default. This prevents uncontrolled memory consumption, but also avoids scalability problems caused by concurrently accessing the page cache from multiple cores [96]. uDepot supports accessing storage both via synchronous IO and via asynchronous IO. Synchronous IO is implemented by the uDepot Linux backend (called so because scheduling is left to Linux). Despite its poor performance, this backend allows uDepot to be used by existing applications without modifications. For example, we have implemented a uDepot JNI interface that uses this backend. Its implementation is simple, since most operations directly translate to system calls. For asynchronous and user-space IO, uDepot uses TRT, and can use either SPDK or the kernel Linux AIO facility.

4.7 uDepot server

Embedded uDepot provides two interfaces to users: one where operations take arbitrary (contiguous) user buffers, and one where operations take a data structure that holds a linked list of buffers allocated from uDepot. The former interface, which internally is implemented using the latter, is simpler but is inherently inefficient. One of the problems is that for many IO backends it requires a data copy between IO buffers and the user-provided buffers. For instance, performing direct IO requires aligned buffers, while SPDK requires buffers allocated via its run-time system. Our server uses the second interface so that it can perform IO directly from (to) the receive (send) buffers. The server is implemented using TRT and uses the epoll backend for networking. First, a task for accepting new network connections is spawned. This task registers with the poller, and is notified when a new connection is requested. When this happens, the task will check if it should accept the new connection and spawn a new task on a (randomly chosen) TRT thread. The task will register with the local poller to be notified when there are incoming data for its connection. The connection task handles incoming requests by issuing IO operations to the storage backend (either Linux AIO or SPDK). After issuing an IO request, the task defers its execution and the scheduler runs another task. The storage poller is responsible for waking up the deferred task when the IO completion is available. The task will then send the proper reply and wait for a new request.

4.8 Memcache server

uDepot also implements the Memcache protocol [64], widely used to accelerate object retrieval from slower data stores (e.g., databases). The standard implementation of Memcache is in DRAM [65], but implementations for SSDs also exist [27, 61].

uDepot Memcache is implemented similarly to the uDepot server (§4.7): it avoids data copies, uses the epoll backend for networking and either the AIO or the SPDK backend for access to storage. Memcache specific KV metadata (e.g., expiration time, flags, etc.) are appended at the end of the value. Expiration is implemented in a lazy fashion: it is checked when a lookup is performed (either for a Memcache GET or a STORE command).

uDepot Memcache exploits synergies in the cache eviction and the space management GC design space: a merged cache eviction and GC process is implemented that reduces the GC cleanup overhead to zero in terms of IO amplification. Specifically, a GC LRU-policy is employed at the segment level (§4.1): on a cache hit the segment containing the KV is updated as the most recently accessed; when running low on free segments the least recently used one is chosen for cleanup, its valid KV entries (both expired and unexpired) are invalidated (i.e., evicted) in the uDepot directory, and the segment is now free to be re-filled, without performing any relocation IO. This scheme allows us to maintain a steady performance even in the presence of sustained random updates, and also to reduce the overprovisioning at the space management level (SALSA) to a bare minimum (enough spare segments to accommodate the supported write-streams) thus maximizing capacity utilization at the space management level. A drawback of this scheme is potentially reduced cache hit ratio [81,93]; we think this is a good tradeoff to make since the cache hit ratio is amortized by having a larger caching capacity due to the reduced overprovisioning. The uDepot memcache server is the basis of an experimental cloud memcache service currently available in the public cloud [39].

4.9 Implementation notes

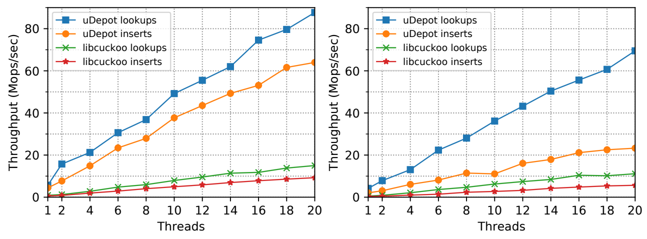
uDepot is implemented in C++11. It is worth noting that uDepot's performance requires many optimizations: we eliminate heap allocations from the data path using core-local slab allocators, we use huge pages, we favor static over dynamic polymorphism, we avoid copies using scatter-gather IO and placing data from the network at the proper location of IO buffers, we use batching, etc.

5 Evaluation

We perform our experiments on a machine with two 10-core Xeon CPUs (configured to operate at their maximum frequency: 2.2GHz), 125 GiB RAM, and running a 4.14 Linux kernel (including support for KPTI [16] – a mitigation for CPU security problems that increases context switch overhead). The machine has 26 NVMe drives: 2 Intel Optanes (P4800X 375GB HHHL PCIe), and 24 Intel Flash SSDs (P3600 400GB 2.5in PCIe).

5.1 Index structure

We start by evaluating the performance of our index structure both in the absence and presence of resize operations. We use 512 MiB (2^{26} entries) hash tables with 8192 locks per table. Our experiment consists of inserting a number of random keys, and then performing random lookups on those keys. We consider two cases: i) inserting 50M ($5 \cdot 10^7$) items where no resize happens, and ii) inserting 1B (10^9) items where four grow operations happen. We compare against libcuckoo [53, 54], a-state-of-the-art hash table implementation by running its accompanying benchmarking tool (`universal_benchmark`), configuring an initial capacity of $2^{26}/2^{30}$ for our 50M/1B runs. Results are shown in Fig. 5. For 50M items, our implementation achieves 87.7 million lookups and 64 million insertions per second, $\times 5.8$ and $\times 6.9$ better than libcuckoo, respectively. For 1B items, the insertion rate drops to 23.3 Mops/sec due to the resizing



(a) throughput: 50M items (no grow) (b) throughput: 1B items (4 grows)

percentile	lookup/50M	lookup/1B	insert/50M	insert/1B
50%	0.2 µs	0.3 µs	0.2 µs	0.4 µs
99%	1.1 µs	1.2 µs	0.6 µs	1.0 µs
99.9%	1.9 µs	2.0 µs	1.6 µs	9.2 µs
99.99%	11.0 µs	8.9 µs	7.5 µs	1168.0 µs

(c) Operation latencies

Figure 5: Mapping structure performance results.

operations. To better understand the cost of resizing, we perform another run where we sample latencies. Fig. 5c shows the resulting median and tail latencies. The latency of insert operations needing to copy items is seen in the 99.99% percentile, where latency is 1.17 ms. Note that this is a worse case scenario, where only insertions and no lookups are performed. It is possible to reduce the latency of these slow insertions by increasing the number of locks, at the cost of additional memory.

5.2 Embedded uDepot

Next, we examine the performance of uDepot as an embedded store. Our goal is to evaluate uDepot's ability to utilize FNDs, and compare the performance of the three different IO backends: syncronous IO using threads (`linux-directIO`), TRT using Linux asynchronous IO (`trt-aio`), and TRT using SPDK (`trt-spdk`). We are interested in two properties: *efficiency* and *scalability*. For the first, we restrain the application to use 1 core and 1 drive (§5.2.1). For the second, we use 24 drives and 20 cores (§5.2.2).

We use a custom microbenchmark to generate load for uDepot. We annotate the microbenchmark to sample the execution time for the operations performed, which we use to compute the median latency. In the following experiments, we use random keys of 8-32 bytes and values of 4K bytes. We perform 50M random `PUT`s, and 50M random `GET`s on the inserted keys.

5.2.1 Efficiency (one drive, one core)

We evaluate the efficiency of uDepot and its IO backends by using *one* core to drive *one* Optane drive. We compare uDepot's performance to the raw performance achievable by the device.

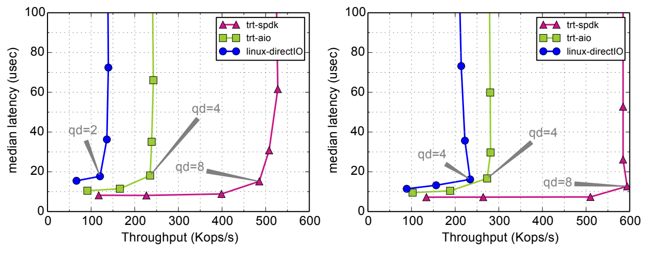


Figure 6: uDepot running on a single core/single device setup. Median latency and throughput for a uniform random workload of 4K values for different IO backends and different queue depths.

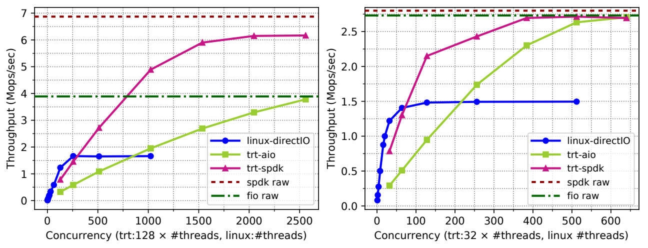


Figure 8: Aggregate GET/PUT throughput of uDepot backends when using 24 NVMe drives for different concurrencies.

We bind all threads on a single core (one that is on the same NUMA node as the drive). We apply the workload described in §5.2 for queue depths (qd) of $1, 2, 4, \ldots, 128$ and for the three different IO backends. For synchronous IO (`linux-directIO`) we spawn a number of threads equal to the qd. For TRT backends we spawn a single thread and a number of tasks equal to the qd. Both `linux-directIO` and `trt-aio` use direct IO to bypass the page cache.

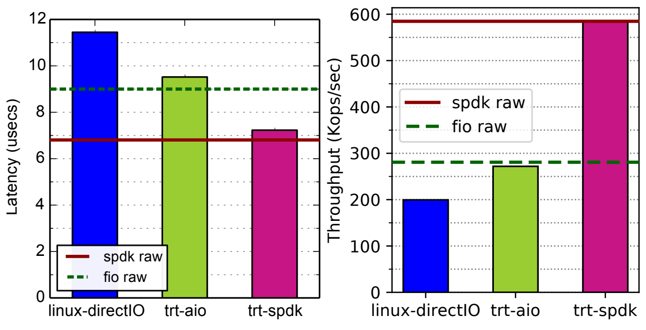


Figure 7: uDepot running on a single core/single device setup under a uniform random workload of GET operations for 4K values.

Results are shown in Fig. 6b for GETs and Fig. 6a for PUTs. The `linux-directIO` backend performs the worst. To a large extent, this is because it uses one thread per in-flight request, resulting in frequent context switches by the OS to allow all these threads to run on a single core. `trt-aio` improves performance by using TRT's tasks to perform asynchronous IO and perform a single system call for multiple operations. Finally, `trt-spdk` exhibits (as expected) the best performance as it avoids switching to the kernel.

We consider the better performing GET operations to compare uDepot against the device performance. We focus on latency with a single request in flight (qd = 1), and throughput at a high queue depth (qd = 128). Fig. 7a shows the median latency achieved for qd = 1 for each backend. The figure includes two lines depicting the raw performance of the device under a similar workload obtained using appropriate benchmarks for each IO facility. That is, one core, one device, 4KiB random READ operations at qd = 1 across the whole device which was randomly written (preconditioned). `fio raw` shows the latency achieved by `fio` [23] with the `libaio` (i.e., Linux AIO) backend, while for `spdk raw` we use SPDK's `perf` utility [86]. uDepot under `trt-spdk` achieves a latency of 7.2 µs which is very close the latency of the raw device using SPDK (6.8 µs). The `trt-aio` backend achieves a latency of 9.5 µs with the corresponding raw device number using fio being 9 µs. An initial implementation of the `trt-aio` backend that used the `io_getevents()` system call to receive IO completions, resulted in a higher latency (close to 12 µs). We improved performance by implementing this functionality in user-space [17, 28, 77]. fio's latency remained unchanged when using this technique (fio option `userspace_reap`). Fig. 7b shows the throughput achieved by each backend at high (128) queue depth. `linux-directIO` achieves 200 kops/s, `trt-aio` 272 kops/s, and `trt-spdk` 585 kops/s. As before, `fio raw` and `spdk raw` show the device performance under a similar workload (4KiB random READs, qd=128) as reported by `fio` and SPDK's `perf`. Overall, uDepot performance is very close to the device performance.

5.2.2 Scalability (24 drives, 20 cores)

Next, we examine how well uDepot scales when using multiple drives and multiple cores, and how the different IO backends behave under these circumstances.

To maximize aggregate throughput, we use the 24 Flash-based NVMe drives in the system, and all of its 20 cores. (Even though these drives are not FNDs, we use a large number of them to achieve a high aggregate throughput and examine uDepot's scalability.) For the uDepot IO backends that operate on a block device (`linux-directIO` and `trt-aio`), we create a software RAID-0 device that combines the 24 drives into one using the Linux `md` driver. For the `trt-spdk` backend we use the RAID-0 uDepot SPDK backend. We use the workload described in §5.2, and take measurements for different numbers of concurrent requests.

For `linux-directIO` we use one thread per request, up to 1024 threads. For TRT backends, we use 128 TRT tasks per thread for `GET`s and 32 TRT tasks for `PUT`s (we use different numbers for different operations because they are saturated at different queue depths). We vary the number of threads from 1 up to 20.

Results are presented in Fig. 8. We also include two lines depicting the maximum aggregate throughput achieved on the same drives by SPDK `perf` and `fio` using the `libaio` (Linux AIO) backend. We focus on `GET`s, because that's the most challenging workload. The `linux-directIO` backend initially has better throughput as it uses more cores. For example, for a concurrency of 256, it uses 256 threads, and subsequently all the cores of the machine; for the TRT backends, the same concurrency uses 2 threads (128 tasks per thread), and subsequently 2 out of the 20 cores of the machine. Its performance, however, is capped at 1.66 Mops/s. The `trt-aio` backend achieves a maximum throughput of 3.78 Mops/s, which is very close to the performance achieved by `fio`: 3.89 Mops/s. Finally, `trt-spdk` achieves 6.17 Mops/s which is about 90% of the raw SPDK performance (6.87 Mops/s). We use normal SSDs to reach a larger throughput than the one that we could using Optane drives due to limited PCIe slots on our server. Because we measure throughput, these results can be generalized to FNDs with the difference being that it would require fewer drives to reach the achieved throughput. Moreover, the raw SPDK performance measured (6.87 Mops/s) is close to the throughput that the IO subsystem of our server can deliver: 6.91 Mops/s. The latter number is the throughput achieved by the SPDK benchmark when using uninitialized drives that return zeroes *without* accessing Flash. The PCIe bandwidth of our server is 30.8 GB/s (or 7.7 Mops/s for 4 KiB), which is consistent with our results if we consider PCIe and other overheads.

Overall, both uDepot backends (`trt-aio`, `trt-spdk`) perform very close in terms of efficiency and scalability to what the device can provide for each different IO facility. In contrast, using blocking system calls (`linux-directIO`) and multiple threads has significant performance limitations both in terms of throughput and latency.

5.3 uDepot server / YCSB

In this section we evaluate the performance of the uDepot server against two NoSQL stores: Aerospike [2] and ScyllaDB [82]. Even though uDepot has (by design) less functionality than these systems, we select them because they are NVMe-optimized and offer, to the best of our knowledge, the best options for exploiting FNDs today.

To facilitate a fair comparison, we use the YCSB [15] benchmark, and run the following workloads: A (update heavy: 50/50), B (read mostly: 95/5), C (read only), D (read latest), and F (read-modify-write), with 10M records and the default record size of 1 KiB. (We exclude workload E because uDepot does not support range queries.) We configure all systems to use two Optane drives and 10 cores (more than enough to drive 2 Optane drives), and generate load using a single client machine connected via 10 Gbit/sec Ethernet. For uDepot, we develop a YCSB driver using the uDepot JNI interface to act as a client. Because TRT is incompatible with the JVM, clients use the Linux uDepot backend. For Aerospike and ScyllaDB we use their available YCSB driver. We use YCSB version 0.14, Scylla version 2.0.2, and Aerospike version 3.15.1.4. For Scylla, we set the cassandra-cql driver's `core` and `maxconnections` parameters at least equal to the YCSB client threads, and capped its memory use to 64GiB to mitigate failing YCSB runs on high client thread counts due to memory allocation.

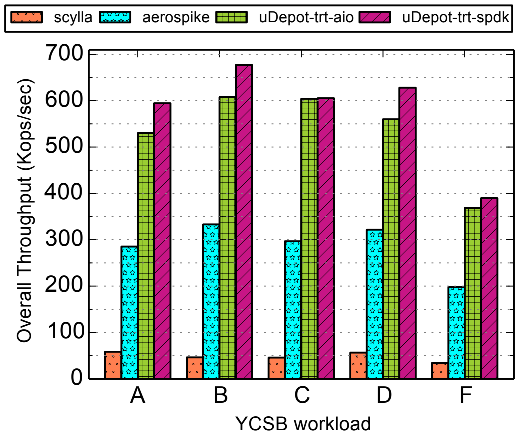


Figure 9: Overall throughput when using 256 YCSB client threads for different key-value stores.

Fig. 9 presents the achieved throughput for 256 client threads for all workloads. uDepot using the `trt-spdk` backend improves YCSB throughput from $\times 1.95$ (workload D) up to $\times 2.1$ (workload A) against Aerospike, and from $\times 10.2$ (workload A) up to $\times 14.7$ (workload B) against ScyllaDB. Fig. 10 focuses on the update-heavy workload A (50/50), depicting the reported aggregate throughput, update and read latency for different number of client threads (up to 256) for all the examined stores. For 256 clients, uDepot using SPDK achieves a read/write latency of 345 µs/467 µs, Aerospike 882 µs/855 µs, and ScyllaDB 4940 µs (3777 µs).

We profile execution under workload A, to understand the causes of the performance differences between Aerospike, ScyllaDB, and uDepot. Aerospike is limited by its use of multiple IO threads and synchronous IO. Indeed, synchronization functions occupied a significant amount of its execution time due to contention created by the multiple threads. ScyllaDB uses asynchronous IO (and in general has an efficient IO subsystem), but it exhibits significant IO amplification. We measured the read IO amplification of the user data (YCSB key and value) versus what was read from the FNDs

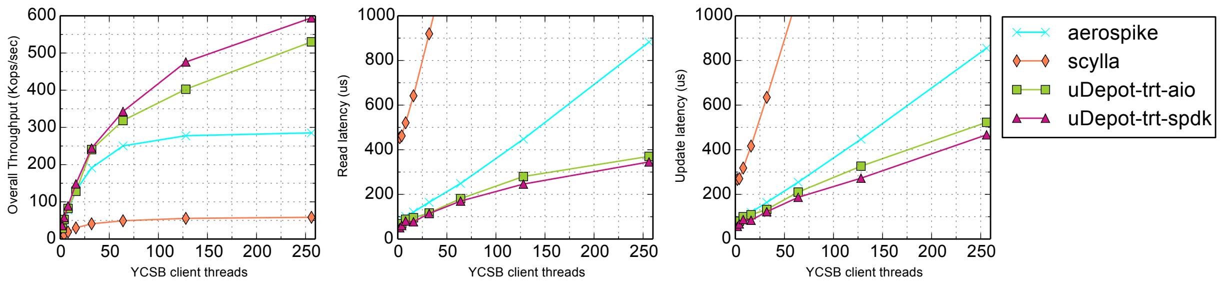


Figure 10: Overall throughput, update and read latency, as reported by the YCSB benchmark for different number of client threads applying workload A (50/50 reads/writes) to different key-value stores.

and the results were as follows: ScyllaDB: ×8.5, Aerospike: ×2.4, and uDepot (TRT-aio): ×1.5.

Overall, uDepot exposes the performance of FNDs significantly better than Aerospike and ScyllaDB. We note that YCSB is inefficient since it uses synchronizing Java threads with synchronous IO, and under-represents uDepot's performance. In the next section, we use a more performant benchmark that better illustrates uDepot's efficiency.

5.4 uDepot Memcache

Lastly, we evaluate the performance of our uDepot Memcache implementation, and investigate if it can provide comparable performance to DRAM-based services.

We use memcached [65] (version: 1.5.4), the standard implementation of Memcache that uses DRAM, as the standard on what applications using Memcache expect, MemC3 [25] (commit: `84475d1`), a state-of-the-art Memcache implementation, and Fatcache [27] (commit: `512caf3`), a Memcache implementation on SSDs.

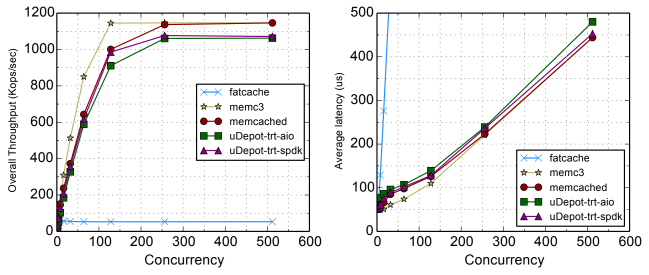


Figure 11: Memcache performance as reported by memaslap using the default 10%-`PUT`, 90%-`GET` workload of 1 KiB objects for different number of clients (concurrency).

We use memaslap[4] [62], a standard Memcache benchmark, and generate the default workload: 10%-`PUT`, 90%-`GET` with 1 KiB objects. We execute memaslap on a different machine, connected over 10 Gbit/s Ethernet to the server. The Memcache servers are configured to use all 20 cores of our machine. DRAM-based memcached, and MemC3 are configured to use enough memory to fit all the working set, while Fatcache and uDepot are configured to use the two Optane drives in a RAID-0 configuration, using the Linux md driver when required. We use the default options for Fatcache.

[4]We applied a number of scalability patches [63] to improve performance.

The reported latency and throughput is summarized in Fig. 11. For a single client, the reported latency is 49 µs for MemC3, 51 µs for both `memcached` and uDepot using `trt-spdk`, 52 µs for Fatcache, and 67 µs for uDepot using `trt-aio`. Contrarily to uDepot, Fatcache caches data in DRAM which leads to the low latency at low queue depths. As the number of clients increase, however, the performance of Fatcache significantly diverges, while uDepot's performance remains close. Case in point, for 128 clients, MemC3's latency is 110 µs, memcached's 126 µs, uDepot with `trt-spdk` achieves 128 µs, uDepot with `trt-aio` 139 µs, and Fatcache 2418 µs; The achieved throughputs are: MemC3:1145 kops/s, memcached:1001 kops/s, uDepot `trt-spdk`: 985 kops/s uDepot `trt-aio`: 911 kops/s, and Fatcache: 53 kops/s.

Hence, our results show that `memcached` on DRAM can be replaced with uDepot on NVM with a negligible performance hit, since the bottleneck is the network. Moreover, Fatcache cannot exploit the performance benefits of FNDs.

6 Related work

Flash KV stores Two early KV stores that specifically targeted Flash are FAWN [3], a distributed KV store, built with low-power CPUs and small amounts of Flash storage, and FlashStore [19], a multi-tiered KV store using both DRAM, Flash, and Disks. These systems are similar to uDepot in that they keep an index in the form of a hash-table in DRAM, and they use a log-structured approach. They both use 6-byte entries: 4 bytes to address Flash, and 2 bytes for they key fingerprint, while subsequent evolutions of these works [20,56] further reduce the entry size. uDepot increases the entry to 8 bytes, enabling features not supported by the above systems: i) uDepot stores the size of the KV entry, allowing it to fetch both key and value with a single read request. That is, a GET operation requires a single access. ii) uDepot supports online resizing that does not require accessing NVM storage. iii) uDepot uses 40 instead of 32 bits for addressing storage, supporting up to 1 PB of grains. Moreover, uDepot efficiently accesses FNDs (via asynchronous IO backends) and scales over many devices and cores which these systems, built for slower devices, do not support. A number of works [60,91] built Flash KV stores or caches [81,84] that rely on non-standard storage devices, such as open-channel SSDs. uDepot does not depend on special devices, and using richer storage interfaces to improve uDepot is future work.

High-performance DRAM KV stores A large number of works targets to maximize the performance of DRAM-based KV stores using RDMA [21,44,68,73], direct access to network hardware [57], or, FPGAs [10, 52]. uDepot, on the other hand, operates over TCP/IP and places data in storage devices. Nevertheless, many of these systems use a hashtable to maintain their mapping, and access it with one-sided RDMA operations from the client when possible. FaRM [21], for example, identifies the problems of cuckoo hashing, and, similarly to uDepot, uses a variant of hopscotch hashing. A fundamental difference of FaRM and uDepot is that the former is concerned with minimizing RDMA operations to access the hash table, which is not a concern for uDepot. Moreover, uDepot's index structure supports online resizing, while FaRM uses an overflow chain per bucket that can cause a performance hit for checking the chain.

NVM KV stores A number of recent works [4,71,92,95] propose NVM KV stores. These systems are fundamentally different in that they operate on byte-addressable NVM placed on the memory bus. uDepot, instead, uses NVM on storage devices because the technology is widely available and more cost effective. MyNVM [22] also uses NVM storage as a way to reduce the memory footprint of RocksDB, where NVM storage is introduced as a second level block cache. uDepot takes a different approach by building a KV store that places data exclusively on NVM. Aerospike [87], that targets NVMe SSDs, follows a similar design to uDepot by keeping its index in DRAM and the data in a log that resides in storage. Nevertheless, because it is designed with SSDs in mind, it cannot fully exploit the performance of FNDs (e.g., it uses synchronous IO). Faster [11] is a recent KV store that, similarly to uDepot, maintains an resizable in-memory hash index and stores its data into a log. In contrast to uDepot, Faster uses a hybrid log that resides both in DRAM and in storage.

Memcache Memcache is an extensively used service [5, 6, 32, 65, 70]. MemC3 [25] redesigns memcached using a concurrent cuckoo hashing table. Similarly to the original memcached, the hash table cannot be dynamically resized and the amount of used memory must be defined when the service starts. uDepot supports online resizing of the hash table, while also allowing for faster warm-up times if the service restarts since the data are stored in persistent storage. Recently, usage of FNDs in memcached was explored as means to reduce costs and expand the cache [66].

Task-based asynchronous IO A long-standing debate exists on programming asynchronous IO using threads versus events [1, 18, 49, 51, 90]. uDepot is built on TRT that uses a task-based approach, where each task has its own stack. A useful extension to TRT would be to provide a composable interface for asynchronous IO [36]. Flashgraph [97] uses an asynchronous task-based IO system to process graphs stored on Flash. Seastar [83], the run-time used by ScyllaDB, follows the same design principles as TRT, but does not (currently) support SPDK.

7 Conclusion and Future Work

We presented uDepot, a KV store that fully utilizes the performance of fast NVM storage devices like Intel Optane. We showed that uDepot reaches the performance available from the underlying IO facility it uses, and can better utilize these new devices compared to existing systems. Moreover, we showed that uDepot can use these devices to implement a cache service that achieves a similar performance to DRAM implementations, at a much lower cost. Indeed, we use our uDepot Memcache implementation as the basis of an experimental public cloud service [39].

uDepot has two main limitations that we plan to address in future work. First, uDepot does not (efficiently) support a number operations that have been proven useful for applications such as range queries, transactions, checkpoints, data structure abstractions [78], etc. Second, there are many opportunities to improve efficiency by supporting multiple tenants [13], that uDepot does not currently exploit.

8 Acknowledgements

We would like to thank the anonymous reviewers and especially our shepherd, Peter Macko, for their valuable feedback and suggestions, as well as Radu Stoica for providing feedback on early drafts of our paper. Finally, we would like to thank Intel for providing early access to an Optane testbed.

References

[1] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative task management without manual stack management. USENIX ATC '02.

[2] Aerospike — high performance NoSQL database. https://www.aerospike.com/.

[3] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A fast array of wimpy nodes. SOSP '09.

[4] Arulraj, J., Levandoski, J., Minhas, U. F., and Larson, P.-A. Bztree: a high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow. 11*, 5 (2018).

[5] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. SIGMETRICS '12.

[6] Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[7] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (2017).

[8] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[9] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. OSDI '14.

[10] Chalamalasetti, S. R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P., and Margala, M. An FPGA memcached appliance. FPGA '13.

[11] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. FASTER: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment* (2018).

[12] Chang, L.-P., Kuo, T.-W., and Lo, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. 3*, 4 (2004).

[13] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. USENIX ATC '17.

[14] CityHash, a family of hash functions for strings. https://github.com/google/cityhash.

[15] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. SoCC '10.

[16] Corbet, J. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, Dec. 2017.

[17] Corbet, J. A new kernel polling interface. https://lwn.net/Articles/743714/, 2018.

[18] Dabek, F., Zeldovich, N., Kaashoek, F., Mazieres, D., and Morris, R. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002).

[19] Debnath, B., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow. 3*, 1-2 (2010).

[20] Debnath, B., Sengupta, S., and Li, J. Skimpystash: RAM space skimpy key-value store on flash-based storage. SIGMOD '11.

[21] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: fast remote memory. NSDI '14.

[22] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM footprint with NVM in Facebook. EuroSys '18.

[23] Flexible I/O tester, https://linux.die.net/man/1/fio.

[24] Facebook. RocksDB — a persistent key-value store. http://rocksdb.org.

[25] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. NSDI '13.

[26] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. SAP HANA database: Data management for modern business applications. *SIGMOD Rec. 40*, 4 (2012).

[27] fatcache. https://github.com/twitter/fatcache.

[28] fio `user_io_getevents()` implementation. https://github.com/axboe/fio/blob/fio-3.3/engines/libaio.c#L120.

[29] C++ documentation: std::future. http://en.cppreference.com/w/cpp/thread/future.

[30] Java documentation: java.util.concurrent: Future. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html.

[31] Google. LevelDB. https://github.com/google/leveldb.

[32] App engine memcache service. https://cloud.google.com/appengine/docs/standard/python/memcache/.

[33] Graefe, G., Volos, H., Kimura, H., Kuno, H., Tucek, J., Lillibridge, M., and Veitch, A. In-memory performance for big data. *Proc. VLDB Endow. 8*, 1 (2014).

[34] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of nand flash memory. FAST '12.

[35] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. OLTP through the looking glass, and what we found there. SIGMOD '08.

[36] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. AC: Composable asynchronous io for native languages. OOPSLA '11.

[37] Herlihy, M., Shavit, N., and Tzafrir, M. Hopscotch hashing. DISC '08.

[38] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (1989).

[39] "IBM". Data store for memcache. https://cloud.ibm.com/catalog/services/data-store-for-memcache.

[40] Are the major dram suppliers stunting dram demand? http://www.icinsights.com/news/bulletins/Are-The-Major-DRAM-Suppliers-Stunting-DRAM-Demand/. Accessed: 2018-09-10.

[41] Ioannou, N., Kourtis, K., and Koltsidas, I. Elevating commodity storage with the SALSA host translation layer. MASCOTS '18.

[42] io_getevents(2) - read asynchronous i/o events from the completion queue. http://man7.org/linux/man-pages/man2/io_getevents.2.html.

[43] io_submit(2) - submit asynchronous I/O blocks for processing. http://man7.org/linux/man-pages/man2/io_submit.2.html.

[44] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. USENIX ATC '16.

[45] Kegel, D. The c10k problem. http://www.kegel.com/c10k.html, 2014.

[46] Kerrisk, M. *The Linux Programming interface.* 2010.

[47] Kim, H.-J., Lee, Y.-S., and Kim, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. HotStorage '16.

[48] Koh, S., Lee, C., Kwon, M., and Jung, M. Exploring system challenges of ultra-low latency solid state drives. HotStorage '18.

[49] Krohn, M., Kohler, E., and Kaashoek, M. F. Events can make sense. USENIX ATC '07.

[50] Kyoto cabinet: a straightforward implementation of dbm. http://fallabs.com/kyotocabinet/, 2011.

[51] Lauer, H. C., and Needham, R. M. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev. 13*, 2 (1979).

[52] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. KV-Direct: high-performance in-memory key-value store with programmable NIC. SOSP '17.

[53] Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. EuroSys '14.

[54] libcuckoo. https://github.com/efficient/libcuckoo. Accessed: 2018-09-20.

[55] libevent – an event notification library. http://libevent.org/. Accessed: 2017-02-27.

[56] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. SOSP '11.

[57] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. NSDI '14.

[58] Lu, L., Pillai, T. S., Gopalakrishnan, H., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: separating keys from values in SSD-conscious storage. *Trans. Storage 13*, 1 (2017).

[59] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. EuroSys '12.

[60] Marmol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B., and Ganesan, S. NVMKV: A scalable and lightweight flash aware key-value store. HotStorage '14.

[61] Mcdipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/, 2013.

[62] memaslap - Load testing and benchmarking a server. http://docs.libmemcached.org/bin/memaslap.html.

[63] Scalability issues with memaslap client. https://bugs.launchpad.net/libmemcached/+bug/1721048.

[64] Memcache protocol. https://github.com/memcached/memcached/wiki/Protocols. Retrieved Oct 2017.

[65] memcached – a distributed memory object caching system. http://www.memcached.org/.

[66] Caching beyond RAM: the case for NVMe. https://memcached.org/blog/nvm-caching/. Accessed: 2019-12-15.

[67] Menon, J. A performance comparison of RAID-5 and log-structured arrays. In *High Performance Distributed Computing* (1995).

[68] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. USENIX ATC '13.

[69] Mutlu, O., and Subramanian, L. Research problems and opportunities in memory systems. *Supercomput. Front. Innov.: Int. J. 1*, 3 (2014).

[70] 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. NSDI '13.

[71] 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. SIGMOD '16.

[72] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMClouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010).

[73] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud storage system. *ACM Trans. Comput. Syst. 33*, 3 (2015).

[74] Paik, Y. Developing extremely low-latency NVMe SSDs. Flash Memory Summit, 2017. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FA21_Paik.pdf.

[75] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. OSDI '14.

[76] Pletka, R., Koltsidas, I., Ioannou, N., Tomić, S., Papandreou, N., Parnell, T., Pozidis, H., Fry, A., and Fisher, T. Management of next-generation nand flash to achieve enterprise-level endurance and latency targets. *ACM Trans. Storage 14*, 4 (2018).

[77] qemu io_getevents_peek() and io_getevents_commit() implementation. https://git.qemu.org/?p=qemu.git;a=blob;f=block/linux-aio.c;h=88b8d55ec71076e24436ba4a80ec6de4d711e896;hb=HEAD#l131.

[78] Redis. http://redis.io/.

[79] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992).

[80] Rumble, S. M., Kejriwal, A., and Ousterhout, J. Log-structured memory for dram-based storage. FAST '14.

[81] Saxena, M., Swift, M. M., and Zhang, Y. Flashtier: A lightweight, consistent and durable storage cache. EuroSys '12.

[82] ScyllaDB. http://www.scylladb.com/.

[83] Seastar: High performance server-side application framework. http://www.seastar-project.org/. Accessed: 2017-03-01.

[84] Shen, Z., Chen, F., Jia, Y., and Shao, Z. DIDACache: A deep integration of device and application for flash based key-value caching. FAST '17.

[85] Storage performance development kit. http://www.spdk.io/.

[86] Spdk perf. https://github.com/spdk/spdk/blob/master/examples/nvme/perf/perf.c.

[87] Srinivasan, V., Bulkowski, B., Chu, W.-L., Sayyaparaju, S., Gooding, A., Iyer, R., Shinde, A., and Lopatic, T. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.* (2016).

[88] Tallis, B. The intel Optane SSD DC P4800X (375GB) review: Testing 3D XPoint performance. http://www.anandtech.com/show/11209/intel-optane-ssd-dc-p4800x-review-a-deep-dive-into-3d-xpoint-enterprise-performance, 2017.

[89] Tallis, B. Samsung launches Z-SSD SZ985: Up to 800gb of Z-NAND. https://www.anandtech.com/show/12376/samsung-launches-zssd-sz985-up-to-800gb-of-znand, 2018.

[90] von Behren, R., Condit, J., and Brewer, E. Why events are a bad idea (for high-concurrency servers). HOTOS '03.

[91] 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. EuroSys '14.

[92] Xia, F., Jiang, D., Xiong, J., and Sun, N. HiKV: a hybrid index Key-Value store for DRAM-NVM memory systems. USENIX ATC '17.

[93] Xia, Q., and Xiao, W. High-performance and endurable cache management for flash-based read caching. *IEEE Transactions on Parallel and Distributed Systems 27*, 12 (2016).

[94] Yang, J., Minturn, D. B., and Hady, F. When poll is better than interrupt. FAST'12.

[95] 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. FAST '15.

[96] ZHENG, D., BURNS, R., AND SZALAY, A. S. A parallel page cache: Iops and caching for multicore systems. HotStorage '12.

[97] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. FAST '15.