RZ 3923 (# ZUR1802-010) 02/06/2018
Computer Science 14 pages

Research Report

Unveiling the Performance of Fast NVM Storage with the uDepot KV-Store

Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas

IBM Research – Zurich
8803 Rüschlikon
Switzerland
{kou, nio, iko}@zurich.ibm.com

© 2019 The authors

The final version of this paper has been published as "Reaping the performance of fast NVM storage with uDepot," in: Proc.17th USENIX Conference on File and Storage Technologies (FAST '19), Boston, MA, pp. 1-15

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Africa • Almaden • Austin • Australia • Brazil • China • Haifa • India • Ireland • Tokyo • Watson • Zurich

Unveiling the performance of fast NVM storage with the uDepot KV-store

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

Abstract

Storage devices based on novel NVM technologies, such as 3D XPoint, have recently become available and drastically improve performance compared to conventional SSDs. On one hand, these devices present opportunities to accelerate systems by complementing conventional (slower) storage and to reduce the costs of systems that rely on DRAM in order to meet their performance demands. On the other hand, their performance exposes many inefficiencies in applications and the IO stack, rendering them unable to fully exploit these new devices.

In this paper, we present uDepot, a key-value store built bottom-up to match the performance of fast NVM storage devices. uDepot is carefully crafted to avoid inefficiencies, and employs a novel, task-based IO runtime system to maximize IO performance, enabling applications to use fast NVM devices at their full potential. We show that uDepot's performance nearly matches the raw capabilities of the underlying devices, both in terms of throughput and latency, and is highly scalable across multiple devices. As an example, uDepot delivers more than 6 million `GET` operations per second on a single server. Moreover, we demonstrate a memcache service built on top of uDepot that offers similar performance to DRAM-based implementations at a much lower cost.

1 Introduction

Advancements in memory technologies have enabled storage devices with unprecedented performance: they achieve hundreds of thousands of IO operations per second (IOPS) with a few microseconds of latency per IO operation. To give two examples, the Intel Optane SSD, which is based on 3D XPoint (3DXP) memory, improves upon the throughput and latency of conventional Flash-based SSDs by an order of magnitude [75], while Samsung is pioneering new NAND-Flash designs with significantly reduced latency, such as the Z-SSD [63]. Case in point, the latency for reading a single 4KiB block for a conventional Flash-based SSD is in the order of 80 µs, while for Z-SSD it is 12 µs, and for Optane it is 7 µs. Furthermore, the NVM Express (NVMe) [36, 60] protocol provides a standardized, low-latency, streamlined and scalable interface for accessing high-performance storage devices, addressing the performance and efficiency shortcomings of legacy protocols (e.g., SATA) [80].

The emerging fast storage devices sit between conventional SSDs and DRAM in the memory hierarchy and constitute a new and discrete point in the performance/cost-for-capacity tradeoff spectrum. Hence, we view these devices as a counterweight to the architectural trend of placing all data in main memory [23, 30, 61, 62, 66], and aim to use fast NVM storage to build more cost-effective and scalable data stores.

A prominent example of pervasive data stores that make heavy use of DRAM is key-value stores, which are extensively used in modern stacks [21]. Existing key-value stores are unable to fully exploit the performance of these emerging fast storage devices. Key-value stores that place all their data in DRAM [18, 22, 37, 43, 46, 49, 57, 62] either cannot transparently use storage (e.g., systems using RDMA where memory needs to be pinned), or have to rely on OS paging, known to degrade performance [29]. On the other hand, key-value stores that place their data in storage [7, 20, 27, 41], even those that specifically target Flash SSDs [3, 15, 16, 48, 50, 71, 74, 78], are designed with slower storage devices in mind and cannot fully reap the performance of devices such as the Optane drive, both in terms of latency and throughput. For example, most of these systems access storage via blocking system calls, an IO path that underperforms when used with fast NVM devices.

In this paper, we present uDepot (pronounced micro-depo), a key-value store designed from the ground up for fast NVM devices to fill the gap between existing DRAM-based and storage-based systems and operate on the µs scale [6]. By design, uDepot is *lean*: it provides a

streamlined set of functions that enable data access with low-latency, but is not, at its core, burdened with richer functionality. It is also *scalable* as it can nicely scale its performance as one increases the number of SSDs, the number of available CPU cores. Importantly, uDepot is *efficient* in that it a) supports a high throughput per CPU core, b) it achieves a high utilization of storage capacity, c) it carefully enforces low bounds to end-to-end IO amplification, for both reads and writes. The core of uDepot is an embedded store that can be used directly by applications, which we use to build network services on top. One such server uses a custom protocol for key-value access over the network. Another integrates uDepot as a cache and implements the Memcache [54] protocol, allowing it to be used as a drop-in replacement for memcached [55], a widely used [4, 59] DRAM-based cache.

The contributions of this paper are summarized below:

- We recognize the potential of novel NVM-based storage media to replace certain DRAM-hungry applications, such as caches, without performance compromises, enabling dramatic cost reduction.
- We present uDepot, a novel key-value store that targets fast NVM storage, and combines multiple techniques to overcome performance bottlenecks and enable key-value storage with low latency, high CPU and storage efficiency, and high scalability.
- Our experimental evaluation demonstrates that uDepot matches the performance of fast NVM devices, vastly outperforms existing storage-based systems by up to $\times 11.32\%$, but also matches the performance of a memory-backed cache; thus, uDepot can be used as a drop-in replacement for DRAM-based systems to dramatically reduce cost.

The rest of the paper is organized as follows. We provide background and motivation in §2, present and evaluate uDepot in §3 and §4. We discuss related work in §5 and conclude in §6.

2 Background and Motivation

Many storage systems have been architected to store their data in main memory [30, 47, 61]. This trend was motivated by performance gains and facilitated by DRAM scaling economics. With key-value storage becoming increasingly popular in modern stacks [17, 26], researchers have been conducting an "arms race" to maximize performance for in-memory key-value stores [8, 18, 37, 43, 46, 57, 62]. As DRAM memory is approaching physical scaling limits [58] and capacity increase is slowing down, such systems rely on scaling out to achieve the required storage capacity. Naturally, this is inefficient and comes at a high cost as the rest of the node

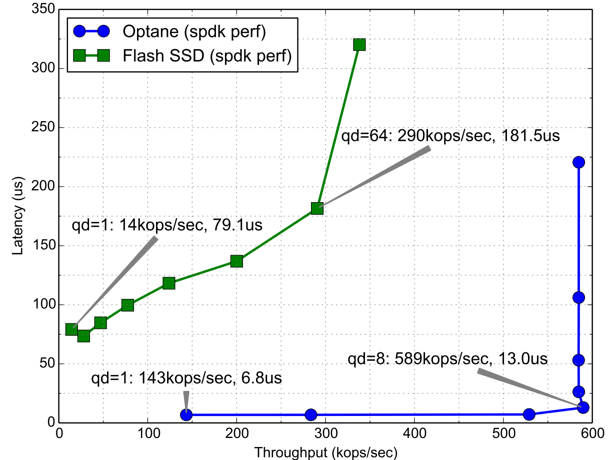


Figure 1: Latency and throughput of 4 KiB random reads on two NVMe devices: a conventional Flash SSD, and an Optane, as we vary the queue depth (ops in flight) in powers of two.

(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 achieved by these systems is impressive, they typically depend on high-performance networking (e.g., RDMA NICs). Thus, they cannot be deployed in commodity datacenter infrastructures such as the ones offered by cloud providers.

Recently, a new class of storage devices built off novel NVM technologies has become available, bridging the gap between DRAM and conventional Flash SSDs as it provides low-latency and high-throughput access. Fig. 1 illustrates the performance differences between an Optane drive and a conventional Flash SSD as measured using SPDK's perf [73]. A single Optane drive, based on 3DXP, can deliver a throughput close to 600 kops/s, and achieve read access latencies of 7 µs (see §4.1.1). Furthermore, Samsung announced availability of a new device [76] that utilizes a new type of Flash (Z-NAND [63]) and has similar performance characteristics to Optane. It is also worth noting that Optane drives are already available in the public cloud [52].

Hence, a key-value store that would deliver the performance of these devices would be an attractive alternative, performing better than existing SSDs at a lower cost than DRAM. Furthermore, in environments with conventional networking (10 Gbit/s Ethernet) such as most cloud platforms, the full performance of DRAM key-value stores cannot be obtained, resulting in a small or marginal performance gap between main memory and fast NVM storage systems. Delivering the performance of these devices in a key-value store (or any application), however, is not easy since existing applications are not built to operate at microsecond scale [6].

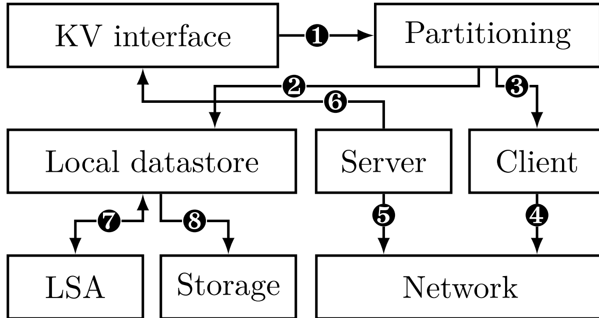


Figure 2: uDepot architecture

3 uDepot

We start with a discussion of the main (in some cases competing) design goals of uDepot. First and foremost, uDepot aims to maximize performance and fully utilize fast NVM devices. To this end, uDepot exposes a minimal interface: `GET`, `PUT`, and `DELETE` operations on variable-sized keys and values. uDepot is build around a hash table allowing fast element access but lacking efficient range queries. Second, uDepot aims to minimize the metadata capacity overhead, allow accessing PBs of storage. This requires having: (a) sufficiently large addresses to storage, and (b) enough hash entries to fit the required number of key-value pairs. Third, uDepot emphasizes efficiently accessing storage using the best available IO facility. Finally, uDepot aims to be memory efficient. It does so by supporting online resizing of the index structure with minimal disruption to operations.

Next, we discuss uDepot in detail. We start with an architectural overview (§3.1) and dive into: space allocation (§3.2), datastore design, (§3.3, §3.4), uDepot operations (§3.5), persistence (§3.6), IO (§3.7), and implementing uDepot network servers (§3.8, §3.9).

3.1 Architecture

uDepot is a distributed key-value server. It partitions the key space using a consistent hash that maps keys to the servers. Each node might act as a client, a server, or both. Fig. 2 shows uDepot's architecture. Operations (e.g., `GET`) first check the consistent hash (❶) to determine whether the operation can be executed locally (❷). If not, the system acts as a client (❸) and forwards the request to one or more uDepot servers (❹). The uDepot server component receives and decodes requests from the network (❺) and then uses the existing key-value interface to serve them (❻). The server and client components are responsible for implementing the network protocol. Currently, uDepot supports a custom uDepot protocol, and the Memcache protocol [54]. This architecture allows different configurations, based on how the partitioning component is implemented. In this paper, we use uDepot as an embedded store where all operations are executed locally (akin to RocksDB [20]), or under a client-server model where clients do not maintain keys and always contact a single server that holds all of the keys (akin to memcahed [55]).

3.2 Space management with SALSA

uDepot manages space using a log-structured approach [56, 67], i.e., space is allocated sequentially and garbage collection (GC) deals with fragmentation. Specifically, we use SALSA [34], running as a user-space library. We use SALSA for a number of reasons. First, it enables good performance on commodity Flash-based SSDs and idiosyncratic storage in general [34]. Second, log-structured allocation is more efficient than traditional allocation methods even for non-idiosyncratic storage like DRAM [68]. Third, a major use case for uDepot is caching, and there are a number of optimization opportunities when co-designing GC and caches [69, 71].

Fully describing SALSA is beyond the scope of this paper. Instead, we present an overview, focusing on aspects that relate to uDepot. SALSA splits the available storage space into *segments*, and segments are, in turn, split into *grains*. SALSA supports multiple "controllers" over a single pool of storage, each with its own policy and allocation streams. A segment can only be owned by a single controller. uDepot implements two SALSA controllers: one for storing a log of key-value records, and one for persisting the index hash tables.

```
alloc_grains(ngrains) -> addr
invalidate_grains(addr, ngrains)
gc_callback(addr, ngrains)
seg_md_callback(addr, ngrains)
seg_iterator()
```

Table 1: SALSA space management interface

The SALSA API is summarized in Table 1. The `alloc_grains` call requests space (in granularity of grains), while `invalidate_grains` is used to inform SALSA that the specified grains are no longer used. SALSA tracks the valid grains in segments, and uses this information to guide GC. To perform relocation, SALSA uses `gc_callback` to upcall its user to relocate the specified grains. When a segment is allocated SALSA upcalls the proper user of the segment using `seg_md_callback` so that they can store their metadata into the segment reserved grain(s) pointed by (addr, ngrains). During startup, users can iterate over SALSA segments using the `seg_iterator()` interface in order to read the segment metadata and restore the segment data as it sees fit.

3.3 Mapping structure

uDepot stores key-value records in NVM storage in a log, while maintaining an in-memory two-level directory for mapping keys to record locations. To speed up restoring the mapping structure, in-memory structures are also saved to persistent storage, but they are not guaranteed to be up-to-date. The persistent source of truth is the log. The directory is an atomic pointer to a read-only array of pointers to hash tables. This structure allows for lock-free accesses when no resize (§3.4) is running.

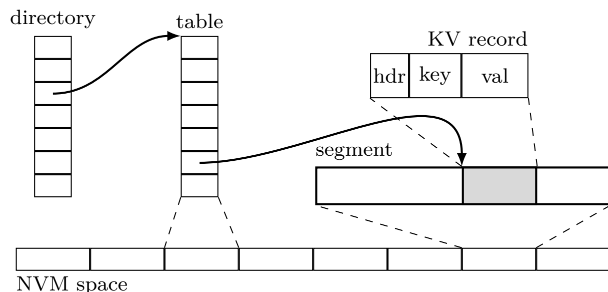


Figure 3: uDepot mapping structure

Each directory entry points to a hash table implementing a modified hopscotch algorithm [32]. We choose hopscotch because of its high density, cache efficient and high performance lookup even in high occupancy, and simple concurrency control [18]. We make three main modifications to the algorithm. First, our version requires a hash table with a power of two number of entries. This allows us to calculate the hopscotch bucket with the least-significant bits (LSB) of the fingerprint; existing entries are thus not required to store those bits but only the distance from their bucket; they are implicit based on the entry location, similarly to the indexing of set-associative caches [33]. We take this approach to have the (full) fingerprint at run-time and support efficient online resize (§3.4), without having to read the keys from storage to recompute their fingerprints. Second, concurrency control is provided through an array of locks. These locks operate on different regions of the hash table, with a region being strictly larger than the bucket size. A lock is acquired based on the bucket's region; if a bucket spans two regions, a second lock is acquired in order. To avoid complex ordering schemes for deadlock prevention, buckets do not wrap-around the hash table. Moreover, to avoid inserts spanning more than two regions, we do not displace entries further than two regions apart. Last, we do not maintain a bitmap per bucket, nor a linked-list of entries per bucket, as in the original variant [32]. A bitmap per bucket would increase our size requirements by 50% for the default configuration (8B entries, and bucket size of 32), and a linked list would at least double it (assuming 8B pointers and singly or doubly linked list); let alone the increase in complexity. A linear probe is performed directly on the entries both for lookup and insert. The raw multi-threaded performance of the hash table at the default configuration (bucket size of 32, 1GiB size with 2^{27} entries, and 2048 locks) on a dual 10-core machine, at 90% occupancy (up-to 90% for inserts), amounts to 4.3M lookups/sec and 2.7M inserts/sec on 1 thread, and 43.4M lookups/sec and 8.9M inserts/sec on 20 threads.

Each table consists of 8-byte entries:

```
struct HashEntry {
  u64 bucket_off:5;  // bucket offset
  u64 key_fp_tag:8;  // fingerprint MSBs
  u64 kv_size:11;  // KV size (grains)
  u64 pba:40;  // storage offset (grains)
};
```

The `pba` field contains the grain offset on storage where the key-value pair resides. To allow utilization of large-capacity devices we use 40 bits for this field, thus able to index petabytes of storage (4 PiB for 4 KiB grains, 0.5 PiB for 512b grains). The `pba` value of all 1s indicates an invalid (free) entry. We use 11 bits to store the size, in grains, of the key-value pair (`kv_size`). This allows issuing a single read for `GET`s to key-value pairs of up-to 8 MiB – key-value pairs larger than that require a second operation. A valid entry with a key-value size of 0 indicates a deleted entry. The remaining 13 bits are used for the key fingerprint as follows. The in-memory structures operate on variable size key fingerprints (up-to 64 bits), computed from the key. A 64 bit hash is generated using cityhash [10], out of which a fingerprint is generated using the $key_fp_tag + idx$ LSB bits. The *idx* bits are equal to the $\log_2(entries)$ of each table. The offset of the entry from its bucket is stored in `bucket_off` and requires $\log_2(bucket_size)$ bits. The remaining bits are used to store the MSB bits of the fingerprint tag in `key_fp_tag` (8 bits for default bucket size).

For *lookups*, a fingerprint is generated, and its LSB bits are used to index the directory and locate the associated hash table. Next, the *idx* LSB bits of the key fingerprint are used to index the bucket in the table. A linear probe is then performed in the bucket and the entry (entries) for which the fingerprint's `key_fp_tag` matches is (are) returned (a copy of), otherwise null.

For *inserts*, the hash table and bucket are indexed as described for the lookup. Then a linear probe is performed on the bucket and if no entry matches the `key_fp_tag`, then insert returns the first free entry, if such exists. The user may then fill the entry. If no free entry exists, then the hash table performs a series of displace attempts to neighbouring buckets until a free entry can be brought into the original bucket. If this fails, a no space error is returned, at which point the user usually triggers a resize operation. If a matching entry (entries)

exists, then insert returns the matching entry (entries): the user then decides whether to update an entry in-place or rather the search for a free entry where they left off.

The number of maximum entries dictates how much of the capacity can be utilized: having enough bits to index PBs of storage allows addressing the available capacity, but alone does not guarantee utility. By having a variable size fingerprint, we allow a variable theoretical maximum of indexable entries equal to $2^{tag_bits+idx_bits+bucket_bits}$. For the default configuration, key-value pairs with an average size of 1 KiB, and hash table efficiency of 90%, this allows utilizing 0.9 PiB ($0.9 \cdot 2^{8+27+5} \cdot 2^{10}$) of storage. Based on the expected workload and available capacity, the user can maximize utilization by configuring the table size accordingly.

If there is no sufficient DRAM for all the mappings, individual tables may be paged in and out. Depending on the IO backend, this can be performed transparently by the OS (using `mmap`) or by custom uDepot code. The tables are flushed to storage in normal shutdown to enable fast restoration of the mapping structure, but also periodically to speed recovery. In case of a crash, the mapping structure information can be reconstructed by the latest table and the subsequent KV records that were appended in the log. Versioning information on each segment allows uDepot to distinguish these KV records and restore the mapping structure.

3.4 Resizing

The in-memory structure may dynamically grow and shrink to adjust memory consumption to the number of items stored. The directory grows in powers of two, so that at any point it holds 2^{m-1+n} entries (m the number of hash tables in the directory, and n the number of entries in each hash table). During a grow operation, new tables are allocated and entries are copied from the old tables to the new. We only need the fingerprint to determine the new locations, so no IO operations are required for growing a table. During resize operations, the data structure becomes read-only: `GET`s are served, but `PUT`s and `DELETE`s are blocked until resizing completes.

We use a synchronization primitive similar to a big reader's lock [64] at the directory level. There are three types of threads based on accesses: readers, writers, and resizers. Each uDepot thread has its own atomic variable, initialized to a large value called the bias (`0x01000000`).[1] Readers and writers take the thread-local lock by decreasing the variable by one. If the value is positive the lock is taken and the operation returns success. Otherwise, the variable is increased by one and failure is returned, indicating that a resize is in progress. Hence, non-resize operations execute concurrently, and because locks are thread-local, there is no contention for non-resizing operations. Resizers are serialized using a single lock. After taking the lock, they allocate a new directory double the size of the existing one, and map the pages of the tables as read-only using `mprotect`. Writers updating pages marked read-only will receive a `SEGFAULT`. uDepot sets signal handlers to handle these signals (if the faults happen on addresses other than the table's, the previously defined action is taken, allowing systems like the JVM to work properly). Before entering the section where such a page fault might occur, readers and writers set a rollback point using `setjmp` [44]. The uDepot `SEGFAULT` handler jumps to the rollback code, where all held locks are released and the thread waits to be woken up by the resizer. Readers continue unobstructed as pages are copied. When copying is complete, the resizer takes *all* the per-thread locks by decreasing the atomic variable by the bias value. At this point, no new readers and writers will be able to enter. The resizer waits until all local-thread variables become zero, at which point it can release the old pages, atomically swap the location of the new directory in place of the old one, free the old directory and continue the execution of readers and writers that are waiting on it.

[1] This variable implements a read-write lock per thread, where uDepot readers and writers take the read lock, and uDepot resizers take the write lock.

3.5 uDepot KV operations

For `GET`, a 64 bit hash of the key is computed and locking of the associated hash table region (§3.3) is performed. A lookup (§3.3) 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 key-value record is fetched 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 key-value 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 (§3.3) 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 (§3.4). 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 key-value 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 that 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 message. `PUT` updates existing entries by default, but provides an optional argument where the user can choose instead to (i) put only if key exists, or (ii) put only if key does not exist.

`DELETE` is almost identical to `PUT`, other than it writes a tombstone entry instead of the key-value record, and that it only proceeds if the key is found in the data store. Tombstone entries are used to identify deleted entries on a restore from the log, and are recycled during GC.

3.6 Metadata and persistence

uDepot maintains metadata at three levels, (i) the device, (ii) the segment, and (iii) the key-value. 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 controller, segment geometry, etc.) together with a timestamp and checksum that matches the device metadata. uDepot prepends to each key-value 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 (0.001% for grain size of 1 KiB and default segment size of 1 GiB). The main overhead is due to the per key value metadata which depends on the average key-value size; for a 1 KiB average size the overhead amounts to 0.8%.

On shutdown, the hash tables(Sec 3.3) are dumped to storage in their respective (allocated at runtime) segments (1 segment per hash table). Upon initialization, we first check whether uDepot was cleanly stopped using per hash table checksums and unique session identifiers. If a clean shutdown is detected, the whole directory is restored. Otherwise, uDepot iterates through the valid segments using SALSA `segment_iterator()`, and restores all key-value entries from valid segments starting at each segment's first grain until it reaches the first non-valid entry (key-value metadata checksum mismatch) or the end of the segment.

3.7 uDepot IO backends (Linux and TRT)

How NVM storage is accessed determines the achieved performance. uDepot by default bypasses the page cache and accesses the storage directly (`O_DIRECT`). This prevents uncontrolled memory consumption, but also avoids scalability problems caused by concurrently accessing the page cache from multiple cores [82]. Broadly speaking, there are three ways to perform IO: (i) via synchronous system calls (e.g., `pread`, `pwrite`), where handling concurrent requests requires one thread for each, leading to context switches that hurt performance. (ii) using asynchronous IO (e.g., Linux AIO [35]), which allows multiple IO requests (and their completions) to be issued (and received) in batches from a single thread. (iii) directly from user-space using polling [38, 72, 81]. This approach provides the best performance because it avoids context switches, data copying, and scheduling overheads, but many environments (e.g., cloud VMs) do not (yet) support it. uDepot supports all above accesses.

IO via synchronous system calls 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, the uDepot JNI interface uses this backend. Its implementation is simple, since most operations directly translate to system calls.

Utilizing fast devices requires performing IO asynchronously. For asynchronous IO, uDepot uses TRT, a task-based run-time system. Next, we provide an overview of TRT (space limitations prohibit a full treatment) and discuss how is used by uDepot. TRT follows task-based model, where a task is a collaboratively scheduled (i.e., no preemption) execution context with its own stack. TRT spawns a number of threads and executes a user-space scheduler on each. The scheduler executes in its own stack and uses a round-robin algorithm. Scheduling is collaborative, so tasks need to perform calls that switch to the scheduler stack. An example of such a call is `yield` that instructs the scheduler to schedule the next task. Other calls include spawning tasks, waiting for events, and notifying tasks. Task synchronization, in particular, is based on calls that implement an interface resembling `Futures`.

TRT provides an infrastructure for asynchronous IO. While many existing systems assume a single point for interacting with the underlying IO facilities (e.g., `epoll`), TRT does not, and can use different IO backends at the same time. Each different IO backend implements a poller task that is responsible for polling for events and notifying tasks that to handle these events. To avoid cross-core communication, each core runs its own poller instance. Poller tasks are scheduled by the scheduler as any other task. TRT currently supports four backends: Linux AIO [35], SPDK [72] (single device and RAID-0 multi-device configurations), and epoll.[2] Backends provide a low-level interface that allows tasks to issue requests and wait on pollers for results, and a high-level interface that allows synchronous-looking code. 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.

[2] backends for RDMA and DPDK are being developed

3.8 uDepot server

uDepot provides two interfaces for users: one where operations take arbitrary (contiguous) user buffers, and one where operations take uBuffs, 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. The uDepot server uses the second interface so that it can perform IO directly from (to) the received (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 tasks 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 request by issuing IO operations to the storage backend (either Linux AIO or SPDK). After issuing an IO request, it defers its execution and the scheduler runs another tasks. The storage poller is responsible for waking up the task when the IO completion is available. The task will then send the proper reply and wait for a new request.

3.9 Memcache

uDepot also implements the Memcache (ascii) protocol [54], widely used to accelerate object retrieval from slower data stores (e.g., databases). The standard implementation of Memcache is in DRAM [55], but implementations that also use SSDs exist [24, 51].

uDepot Memcache is implemented similarly to the uDepot server (§3.8): it uses the uBuff KV interface to avoid copying, the epoll backend for networking, end either the AIO or the SPDK backend for access to storage. Memcache specific key-value 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 's 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 (§3.2): on a cache hit the segment containing the key-value is updated as the most recently accessed; when running low on free segments the least recently used one is chosen for cleanup, its valid key-value entries 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 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 [69, 79]; 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.

4 Evaluation

We perform our experiments on a machine with two 10-core Xeon CPUs, configured to operate at their maximum frequency: 2.2GHz. The machine has 125 GiB RAM and runs a 4.14 Linux kernel (including support for KPTI [12] – a mitigation for CPU security problems that increases context switch overhead). The machine has 26 NVMe drives: 2 Intel Optanes (P4800X 375GB), and 24 Intel Flash SSDs (P3600 400GB).

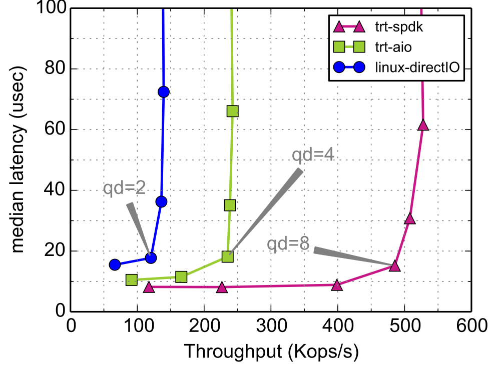
4.1 Embedded uDepot

In this section, we examine the performance of uDepot when used as an embedded store. Our goal is to evaluate uDepot's ability to utilize fast NVM devices, and compare the performance of the different IO backends. We are interested in two properties: *efficiency* and *scalability*. For the first, we restrain the application to use 1 core and 1 drive (§4.1.1). For the second, we use all available drives and cores in the system (§4.1.2).

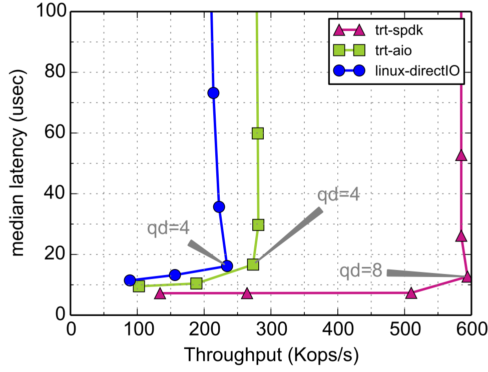
We use a custom microbenchmark to generate load for uDepot. We annotate the microbenchmark to sample the execution time for the operations performed, which we use to compute the median latency. In the following experiments, we use random keys of 8-32 bytes and values of 4000 bytes, which results in key-value entries of 4KiB on the drive (the logical block size of the drives is 512 bytes). We perform 50M random `PUT`s, and 50M random `GET`s on the inserted keys.

4.1.1 Embedded uDepot: 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.

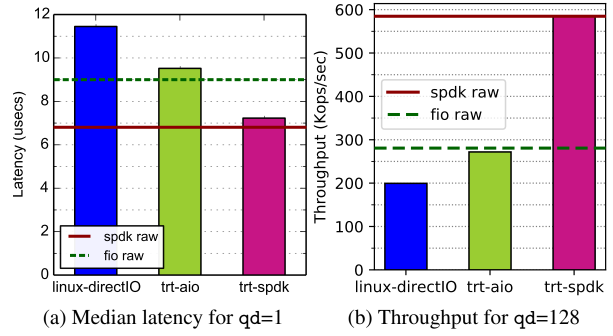


(a) PUTs



(b) GETs

Figure 4: 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.



(a) Median latency for qd=1 (b) Throughput for qd=128

Figure 5: uDepot running on a single core/single device setup under a uniform random workload of `GET` operations for 4K values.

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 §4.1 for queue depths (qd) of 1,2,4,...,128 and for different IO backends. For `linux-directIO` (synchronous IO) 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.

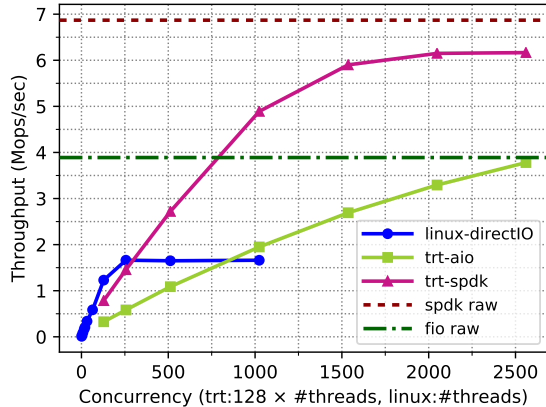
Results are shown in Fig. 4b for `GET`s and Fig. 4a for `PUT`s. The `linux-directIO` backend performs the worse. To a large extent, this is because it uses one thread per in-flight request, resulting in frequent context switches by the OS to allow all these threads to run on a single core. `trt-aio` 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 completely avoids switching to the kernel for performing IO.

We use the better performing `GET` operations to compare uDepot against the performance available from the device. We focus on two cases: latency with a single request in flight (qd = 1), and throughput at a high queue depth (qd = 128). Fig. 5a 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. 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` [19] with the `libaio` (i.e., Linux AIO) backend, while for `spdk raw` we use SPDK's `perf` utility [73]. 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 [13, 25, 65]. fio's latency remained unchanged when using this technique (fio option `userspace_reap`). Fig. 5b shows the throughput achieved by each backend across 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 `READ`s, qd=128) as reported by `fio` and SPDK's `perf`. Raw device performance is virtually the same to uDepot performance.

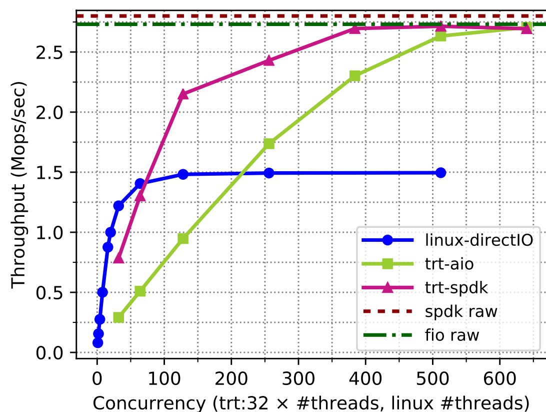
4.1.2 Embedded uDepot: 24 drives, 20 cores

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

To maximize aggregate throughput, the experiment uses the 24 Flash-based NVMe drives in the system, and utilizes all of its 20 cores. For the uDepot IO back-



(a) GETs



(b) PUTs

Figure 6: Aggregate `GET`/`PUT` throughput of uDepot backends when using 24 NVMe drives for different concurrencies.

ends 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 §4.1, 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 (32) TRT tasks per thread for 1,2,4,12,16 for `GET`s (`PUT`s), and 20 threads. (We vary the number of TRT tasks per operation becuse they are saturated at different queue depths.)

Results are presented in Fig. 6. 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, becuase 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 (256 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).

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.

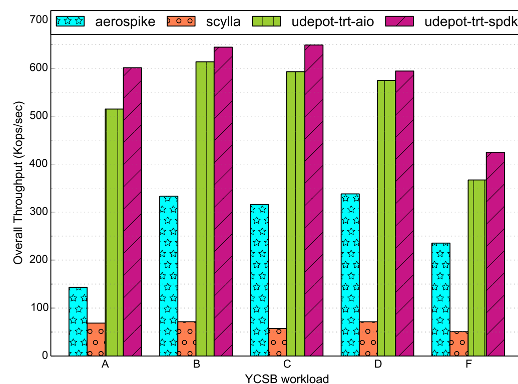


Figure 7: Overall throughput when using 128 YCSB client threads for different key-value stores.

4.2 uDepot server / YCSB

In this section we evaluate the performance of the uDepot server against two state-of-the-art NVMe-optimized NoSQL stores: Aerospike [2] and ScyllaDB [70].

To facilitate a fair comparison, we use the YCSB [11] benchmark. We configure all servers to use two Optane drives. We run both the client and the server for the key-value stores on the same machine using the loopback network interface, dedicating 10 cores to the server and 10 cores to the clients. For uDepot, we develop a YCSB driver using the uDepot JNI interface. Because TRT is incompatible with the JVM, clients use the Linux uDepot backend. For Aerospike and ScyllaDB we use the 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. We run YCSB workloads

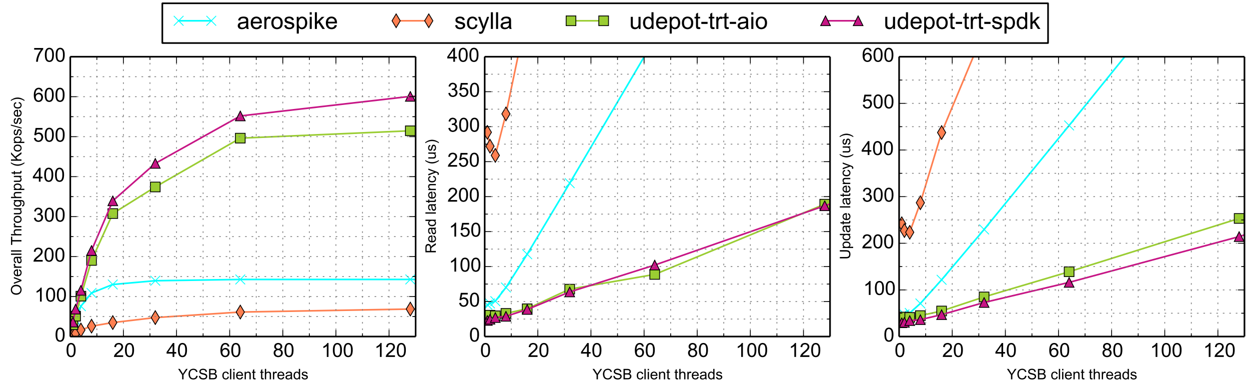


Figure 8: Overall throughput, update and read latency, as reported by the YCSB benchmark for different number of client threads applying workload A (50/50 reads/writes) to different key-value stores.

A,B,C,D,F, with 10M records, and exclude workload E because uDepot does not support range queries.

Fig. 7 presents the achieved throughput for 128 client threads for all workloads. uDepot using the `trt-spdk` backend improves YCSB throughput from ×1.9 (workload B) up to ×4.2 (workload A) against Aerospike, and from ×8.3 (workload D) up to ×11.32 (workload C) against ScyllaDB. 5 Fig. 8 focuses on workload A (50/50 reads and writes), depicting the reported aggregate throughput, update and read latency for different number of client threads (up to 128) for all the examined key-value stores. For 64 clients, uDepot achieves a read (write) latency of 102.2 µs (116.1 µs), Aerospike 425 µs (452.6 µs), and ScyllaDB 1018 µs (1024.4 µs). Overall, uDepot exposes the performance of fast NVMe devices significantly better than Aerospike and ScyllaDB.

4.3 uDepot Memcache

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

We use Memcached [55] (1.5.4), the standard implementation of Memcache that uses DRAM, as the standard on what applications using Memcache expect, MemC3 [22] (commit: `84475d1`), a state-of-the-art Memcache implementation, and Fatcache [24] (commit: `512caf3`), a Memcache implementation on SSDs.

We use memaslap [53][3], a standard Memcache benchmark, to generate workload. We execute memaslap on a different machine (2 10-core HT CPUs running at 2.3GHz) 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

[3] we applied a number of scalability patches [39] to improve performance

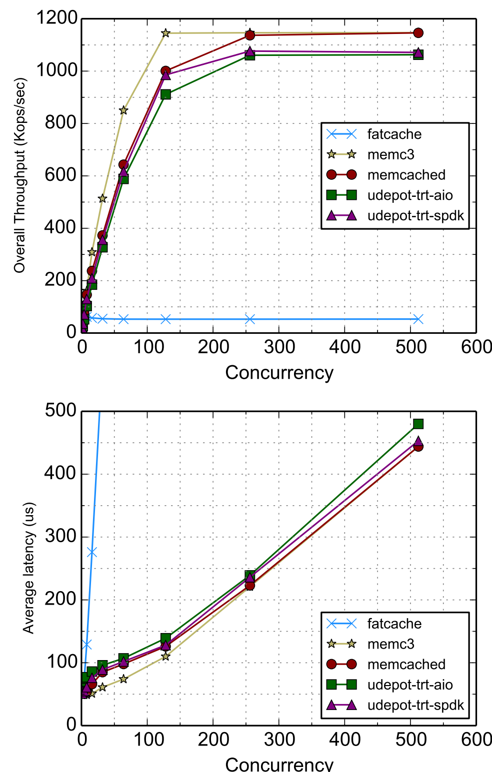


Figure 9: Memcache performance as reported by memaslap using the default 10%-`PUT`, 90%-`GET` workload for different number of clients (concurrency).

the two Optane drives in a RAID-0 configuration, using the Linux md driver when required (other than that, Fatcache uses its default options). We run memaslap using the default workload that generate a 10%-`PUT`, 90%-`GET` workload under different concurrency settings.

The reported latency and throughput is summarized in Fig. 9. For a single client, the reported latency is 49 µs for MemC3, 51 µs for both `memcached` and uDe-

pot 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, for the most common network infrastructure, `memcached` on DRAM can be replaced with uDepot on NVM without any noticeable performance downgrade, while existing approaches for implementing Memcache on SSDs cannot exploit the performance benefits of fast NVM storage.

5 Related work

Flash key-value stores Two key-value stores targeting Flash are FAWN [3], a distributed key-value store, built with low-power CPUs and small amounts of Flash storage, and FlashStore [15], a multi-tiered key-value 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 [16, 45] further reduce the entry size. uDepot is different in that it moves to the opposite direction: it takes advantage of DRAM scaling and increases the entry size to 8 bytes, enabling features not supported by the aforementioned KV-stores: (i) uDepot stores the size of the key-value 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 reading anything from NVM storage. (iii) uDepot uses 40 instead of 32 bits for addressing storage, supporting up to 1 PB of grains. Moreover, uDepot is designed from the ground up for efficient access to fast NVM devices (via different IO backends) and scaling over many devices and cores.

A number of works [50, 78] built Flash key-value stores or caches [69,71] that rely on non-standard storage devices, such as open-channel SSDs. uDepot does not depend on special devices, and using richer interfaces to NVMe storage to improve uDepot is future work.

High-performance DRAM key-value stores A large number of works targets to maximize the performance of DRAM-based key-value stores using RDMA [18, 37, 57, 62], direct access to network hardware [46], or, FPGAs [8, 43]. uDepot, on the other hand, assumes a commodity network infrastructure, operates over sockets and TCP/IP, and places data in NVM storage. Nevertheless, many of these systems use a hash-table to maintain their mapping, and access it with one-sided RDMA operations from the client when possible. FaRM [18], for example, identifies the problems of cuckoo hashing, and, similarly to uDepot, uses a variant of hopscotch hashing. To avoid the cost of RDMA operations, FaRM uses a bucket of 8 entries, while uDepot uses 32. The main difference, however, is that uDepot supports an efficient, online resizing scheme. FaRM avoids resizing by using an overflow chain per bucket which leads to an increased cost for `GET` misses because the chain needs to be checked.

Memcache Memcache is a extensively used service [4, 5, 28, 55, 59]. MemC3 [22] 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. Memshare [9] describes techniques for dynamic sharing across multiple tenants of a memcache service. Currently, multi-tenancy in uDepot is implemented via multiple uDepot services that have a vertical slice of the server. Supporting multiple tenants within uDepot is part of ongoing work.

Task-based asynchronous IO A long-standing debate exists on programming asynchronous IO using threads versus events [1, 14, 40, 42, 77]. 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 [31]. Flashgraph [83] uses an asynchronous task-based IO system to process graphs stored on Flash.

6 Conclusion

We presented uDepot, a key-value store that aims to fully utilize 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.

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. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[4] 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), SIGMETRICS '12.

[5] Amazon elasticache. https://aws.amazon.com/elasticache/.

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

[7] Oracle berkeley db. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[8] Chalamalasetti, S. R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P., and Margala, M. An fpga memcached appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (2013), FPGA '13.

[9] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017).

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

[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), SoCC '10.

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

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

[14] 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).

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

[16] Debnath, B., Sengupta, S., and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), SIGMOD '11.

[17] 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 *ACM SIGOPS operating systems review* (2007), vol. 41.

[18] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014).

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

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

[21] Facebook. RocksDB users. https://github.com/facebook/rocksdb/blob/master/USERS.md.

[22] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[23] 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 (Jan. 2012).

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

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

[26] Fitzpatrick, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004).

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

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

[29] 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 (Sept. 2014).

[30] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), SIGMOD '08.

[31] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. Ac: Composable asynchronous io for native languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2011), OOPSLA '11.

[32] Herlihy, M., Shavit, N., and Tzafrir, M. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing* (2008), DISC '08.

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

[34] Ioannou, N., Kourtis, K., and Koltsidas, I. Elevating commodity storage with the SALSA host translation layer. *ArXiv e-prints* (Jan. 2018). https://arxiv.org/abs/1801.05637.

[35] io_submit(5) - submit asynchronous I/O blocks for processing. http://man7.org/linux/man-pages/man2/io_submit.2.html.

[36] Juenemann, D., and Huffman, A. The nonvolatile memory transformation of client storage. *Computer 46* (2013).

[37] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[38] Kim, H.-J., Lee, Y.-S., and Kim, J.-S. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).

[39] Kourtis, K. Scalability issues with memaslap client. https://bugs.launchpad.net/libmemcached/+bug/1721048.

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

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

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

[43] 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. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM.

[44] Glibc manual: Non-local exits. https://www.gnu.org/software/libc/manual/html_node/Non_002dLocal-Exits.html.

[45] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[46] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014).

[47] Lomet, D., et al. Bulletin of the technical committee on data engineering. *Special Issue on Main-Memory Database Systems 32* (June 2013). http://sites.computer.org/debull/A13june/issue1.htm.

[48] 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 (Mar. 2017).

[49] 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), EuroSys '12.

[50] 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. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (June 2014).

[51] 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.

[52] McNabb, D. Intel Optane SSD DC P4800X available now on IBM cloud. https://www.ibm.com/blogs/bluemix/2017/08/intel-optane-ssd-dc-p4800x-available-now-ibm-cloud/, July 2017.

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

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

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

[56] Menon, J. A performance comparison of raid-5 and log-structured arrays. In *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on* (1995).

[57] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013).

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

[59] 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 *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[60] NVM Express workgroup. *NVM Express*, May 2017. Rev. 1.3.

[61] 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 (Jan. 2010).

[62] 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 (Aug. 2015).

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

[64] Piggin, N. kernel: introduce brlock. https://lwn.net/Articles/378781/, Mar. 2010.

[65] 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.

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

[67] 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).

[68] Rumble, S. M., Kejriwal, A., and Ousterhout, J. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014).

[69] Saxena, M., Swift, M. M., and Zhang, Y. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12.

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

[71] Shen, Z., Chen, F., Jia, Y., and Shao, Z. DIDACache: A deep integration of device and application for flash based key-value caching. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017).

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

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

[74] 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).

[75] 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.

[76] 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, Jan. 2018.

[77] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea (for high-concurrency servers). HOTOS '03.

[78] 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), EuroSys '14.

[79] 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 (Dec 2016).

[80] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), SYSTOR '15.

[81] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12.

[82] ZHENG, D., BURNS, R., AND SZALAY, A. S. A parallel page cache: Iops and caching for multicore systems. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems* (2012).

[83] 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. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Feb. 2015).

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other products and service names might be trademarks of IBM or other companies.