Reaping the performance of fast NVM storage with uDepot

Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas

IBM Research, Zurich

FAST ’19
Many applications require low-latency high-throughput KV storage

- Flash-based SSDs not performant enough
- Most are using DRAM KV-stores (Memcache, Redis)
Many applications require low-latency high-throughput KV storage

- Flash-based SSDs not performant enough
- Most are using DRAM KV-stores (Memcache, Redis)

DRAM performance underutilized on commodity networks (e.g., 10GbE)

- High-performance DRAM KV stores use: RDMA (RamCloud, FaRM), Direct NIC access (MICA), programmable NICs (KV-Direct).
Key-Value (KV) stores

- Many applications require low-latency high-throughput KV storage
  - Flash-based SSDs not performant enough
  - Most are using DRAM KV-stores (Memcache, Redis)

- DRAM performance underutilized on commodity networks (e.g., 10GbE)
  - High-performance DRAM KV stores use: RDMA (RamCloud, FaRM), Direct NIC access (MICA), programmable NICs (KV-Direct).

- DRAM is not getting cheaper
Fast NVM Devices
(FNDs)

- new class of SSDs
  - Intel Optane (3DXP)
  - Samsung Z-SSD (Z-NAND)

- An order of magnitude better performance than Flash SSDs
- Significantly cheaper than DRAM
  - $1.25 vs $10 per GiB (Intel Optane)
  - smaller TCO (number of machines, energy, etc.)
10GiB network vs Flash SSD vs FND SSD

queue depth (reqs in flight): 1, 2, 4, 8, 16, …
10GiB network vs Flash SSD vs FND SSD

- queue depth (reqs in flight): 1, 2, 4, 8, 16, …
- \textit{10GbE}: req:1b, res:4KiB (Intel X710, netperf)

→ Storage no longer the bottleneck!
10GiB network vs Flash SSD vs FND SSD

- queue depth (reqs in flight): 1, 2, 4, 8, 16, …
- 10GbE: req:1b, res:4KiB (Intel X710, netperf)
- SSD: SPDK perf: 4KiB RDs (Flash NVMe)
- Optane: SPDK perf: 4KiB RDs (≈ 0.6Mops/sec, ≈ 10us)

Storage no longer the bottleneck!
10GiB network vs Flash SSD vs FND SSD

- queue depth (reqs in flight): 1, 2, 4, 8, 16, ...
- 10GbE: req:1b, res:4KiB (Intel X710, netperf)
- SSD: SPDK perf: 4KiB RDs (Flash NVMe)
- Optane: SPDK perf: 4KiB RDs ($\approx 0.6$Mops/sec, $\approx 10$us)

→ Storage no longer the bottleneck!
KV store for DRAM FNDs

- reduced cost
- equivalent performance to DRAM KV store (at least, under commodity networks)
KV store for DRAM FNDs

- reduced cost
- equivalent performance to DRAM KV store (at least, under commodity networks)

Existing KV stores cannot deliver FNDs’ performance

- Built for slower devices (e.g., use synchronous IO)
- Data structures with inherent IO amplification (LSM- or B-trees)
- Cache data in DRAM, limiting scalability
- Rich feature set (e.g., transactions, snapshots)

Achieved read throughput on a 20-core 24-device system:

- spdk: 6.87 Mops/sec
- Linux aio: 3.89 Mops/sec
- RocksDB (LSM-tree): 0.96 Mops/sec
- Wiredtiger (B-tree): 0.19 Mops/sec
KV store for DRAM FNDs

- reduced cost
- equivalent performance to DRAM KV store (at least, under commodity networks)

Existing KV stores cannot deliver FNDs’ performance

- Built for slower devices (e.g., use synchronous IO)
- Data structures with inherent IO amplification (LSM- or B-trees)
- Cache data in DRAM, limiting scalability
- Rich feature set (e.g., transactions, snapshots)
Deliver the performance of FNDs to the application

- minimize IO amplification
- scalability (cores, devices, capacity)
- bottom-up approach
  - basic interface: GET, PUT, DEL on variable-sized keys and values.
uDepot architecture
uDepot architecture

Index kept in DRAM
uDepot architecture

Index kept in DRAM

Efficient IO access
uDepot architecture

- Index kept in DRAM
- Efficient IO access
- Log-structured allocation

IBM Research
uDepot architecture

- Index kept in DRAM
- Efficient IO access
- Log-structured allocation

Embeded uDepot
uDepot architecture

Index kept in DRAM

Efficient IO access

Log-structured allocation

Embedded uDepot
uDepot architecture

Index kept in DRAM

Efficient IO access

Log-structured allocation

Embedded uDepot

user app.

memcache server

memcache clients

uDepot server

uDepot clients
uDepot architecture

Index kept in DRAM

Efficient IO access

Log-structured allocation

Embedded uDepot

memcache clients

uDepot clients

user app.

memcache server

uDepot server
IO Facilities

Performance

- Synchronous IO
  - One thread per request
  - Synchronous (blocking) system calls (e.g., `pread`)
  - uDepot Linux backend

- Asynchronous IO
  - Issue IO requests
  - Receive IO completions
  - e.g., Linux AIO
  - uDepot aio backend

- User-space IO
  - Directly access the device
  - Polling instead of interrupts
  - e.g., SPDK
  - uDepot spdk backend

TRT: a run-time system for async IO
Synchronous IO

- one thread per request
- synchronous (blocking) syscalls (e.g., `pread`)

Asynchronous IO

- issue IO requests
- receive IO completions
- e.g., Linux AIO

User-space IO

- Directly access the device
- Polling instead of interrupts
- e.g., SPDK
IO Facilities

Performance

**Synchronous IO**
- one thread per request
- synchronous (blocking) syscalls (e.g., `pread`)

**Asynchronous IO**
- issue IO requests
- receive IO completions
- e.g., Linux AIO

- OS
- Device

TRT: a run-time system for async IO
IO Facilities

- **Synchronous IO**
  - one thread per request
  - synchronous (blocking) syscalls (e.g., `pread`)

- **Asynchronous IO**
  - issue IO requests
  - receive IO completions
  - e.g., Linux AIO

- **User-space IO**
  - Directly access the device
  - Polling instead of interrupts
  - e.g., SPDK

---

TRT: a run-time system for async IO

IBM Research
IO Facilities

Performance

Synchronous IO
- one thread per request
- synchronous (blocking) syscalls (e.g., `pread`)
- uDepot Linux backend

Asynchronous IO
- issue IO requests
- receive IO completions
- e.g., Linux AIO
- uDepot aio backend

User-space IO
- Directly access the device
- Polling instead of interrupts
- e.g., SPDK
- uDepot spdk backend
IO Facilities

Synchronous IO
- one thread per request
- synchronous (blocking) syscall (e.g., `pread`)
- uDepot Linux backend

Asynchronous IO
- issue IO requests
- receive IO completions
- e.g., Linux AIO
- uDepot aio backend

User-space IO
- Directly access the device
- Polling instead of interrupts
- e.g., SPDK
- uDepot spdk backend

TRT: a run-time system for async IO
Goals

- programmer-friendly
  (e.g., avoid stack ripping)
- Handle multiple IO endpoints
  (e.g., SPDK and epoll)
TRT: A task run-time system for asynchronous IO

Goals
▶ programmer-friendly (e.g., avoid stack ripping)
▶ Handle multiple IO endpoints (e.g., SPDK and epoll)

TRT highlights
▶ avoid cross-core communication
TRT: A task run-time system for asynchronous IO

**Goals**

- programmer-friendly
  (e.g., avoid stack ripping)
- Handle multiple IO endpoints
  (e.g., SPDK and epoll)

**TRT highlights**

- avoid cross-core communication
- user-space tasks
  (aka green threads, aka co-routines)
TRT: A task run-time system for asynchronous IO

Goals
▶ programmer-friendly
  (e.g., avoid stack ripping)
▶ Handle multiple IO endpoints
  (e.g., SPDK and epoll)

TRT highlights
▶ avoid cross-core communication
▶ user-space tasks
  (aka green threads, aka co-routines)
▶ poller task for multiple IO backends

Tasks ($T$) issue IO requests
Pollers ($P$) poll for completions
TRT: A task run-time system for asynchronous IO

Goals
- programmer-friendly (e.g., avoid stack ripping)
- Handle multiple IO endpoints (e.g., SPDK and epoll)

TRT highlights
- avoid cross-core communication
- user-space tasks (aka green threads, aka co-routines)
- poller task for multiple IO backends

Tasks (T) issue IO requests
Pollers (P) poll for completions
Pollers notify tasks

Tasks flow diagram:
- Tasks (T) issue IO requests
- Pollers (P) poll for completions
- Pollers notify tasks
uDepot architecture

- Index kept in DRAM
- TRT (Linux AIO, SPDK, Epoll)
- Log-structured allocation

Diagram:
- User app.
- Memcache server
- uDepot server
- IO
- LSA

IBM Research
uDepot architecture

Index kept in DRAM

TRT
(Linux AIO, SPDK, Epoll)

Log-structured allocation

user app.
memcache server
uDepot server
- Two-level hopscotch hash table
  - directory + tables
- 8 byte hash entry (cf. 6-byte for FAWN, FlashStore)
  - maintain KV size in the entry
  - larger storage addresses
- high-performance, scalable
- efficient resizing
Growing the uDepot index

Operations:
- double the size of the directory
- migrate entries to new tables

Minimal disruption
- unobstructed reads
- no IO required: information in the hash entry to reconstruct the fingerprint
- incremental: each operation migrates a bounded number of entries
uDepot architecture

- User app.
- Memcache server
- uDepot server

Two-level resizable hash table

Index

IO

TRT (Linux AIO, SPDK, Epoll)

Log-structured allocation

LSA
uDepot architecture

Two-level resizable hash table

Index

TRT
(Linux AIO, SPDK, Epoll)

SALSA (LSA, GC)

user app.

memcache server

uDepot server

IO

LSA
Evaluation

1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB)
4. uDepot memcache vs DRAM-based memcache implementations
Evaluation

1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB)
4. uDepot memcache vs DRAM-based memcache implementations

Experiment

- vs libcuckoo (better performance, see paper)
- Here: How is tail latency affected by the grow operation?
- ubench: perform 50M (no grow) and 1B (4 grows) inserts and lookups
<table>
<thead>
<tr>
<th>percentile</th>
<th>lookup/50M</th>
<th>lookup/1B</th>
<th>insert/50M</th>
<th>insert/1B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>0.2 µs</td>
<td>0.3 µs</td>
<td>0.2 µs</td>
<td>0.4 µs</td>
</tr>
<tr>
<td>99%</td>
<td>1.1 µs</td>
<td>1.2 µs</td>
<td>0.6 µs</td>
<td>1.0 µs</td>
</tr>
<tr>
<td>99.9%</td>
<td>1.9 µs</td>
<td>2.0 µs</td>
<td>1.6 µs</td>
<td>9.2 µs</td>
</tr>
<tr>
<td>99.99%</td>
<td>11.0 µs</td>
<td>8.9 µs</td>
<td>7.5 µs</td>
<td>1168.0 µs</td>
</tr>
</tbody>
</table>
## uDepot Index Latency

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Lookup/50M</th>
<th>Lookup/1B</th>
<th>Insert/50M</th>
<th>Insert/1B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>0.2 µs</td>
<td>0.3 µs</td>
<td>0.2 µs</td>
<td>0.4 µs</td>
</tr>
<tr>
<td>99%</td>
<td>1.1 µs</td>
<td>1.2 µs</td>
<td>0.6 µs</td>
<td>1.0 µs</td>
</tr>
<tr>
<td>99.9%</td>
<td>1.9 µs</td>
<td>2.0 µs</td>
<td>1.6 µs</td>
<td>9.2 µs</td>
</tr>
<tr>
<td>99.99%</td>
<td>11.0 µs</td>
<td>8.9 µs</td>
<td>7.5 µs</td>
<td>1168.0 µs</td>
</tr>
</tbody>
</table>
Evaluation

1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB)
4. uDepot memcache vs DRAM-based memcache implementations
1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScyllaDB)
4. uDepot memcache vs DRAM-based memcache implementations

Experiment
- uDepot ubench: perform 10M uDepot PUTs and GETs
  - multiple backends (how different backends behave)
- vs. dev ubench: fio and SPDK perf
- same workload: 4KiB
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

LATENCY (µs)

udepot-syncIO udepot-aio udepot-spdk

Throughput (kops)

GET: throughput for qd=128
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

Latency (μs)

udepot-syncIO  udepot-aio  udepot-spdk

Throughput (kops)

GET: throughput for qd=128

IBM Research
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

Latency (µs)

- udepot-syncIO
- udepot-aio
- udepot-spdk

IBM Research
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

Latency (µs)

Latency measurements for different I/O models:
- uDepot-syncIO
- uDepot-aio
- uDepot-spdk

Throughput (kops)

IBM Research
Embedded uDepot: Efficiency (1 core / 1 Optane)

- **GET: median latency for qd=1**

<table>
<thead>
<tr>
<th>Throughput (kops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

- **GET: throughput for qd=128**

<table>
<thead>
<tr>
<th>Throughput (kops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

GET: throughput for qd=128
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

Latency (us)

- udepot-syncIO
- udepot-aio
- udepot-spdk

GET: throughput for qd=128

Throughput (kops)

- udepot-syncIO
- udepot-aio
- udepot-spdk

IBM Research
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

Latency (μs)

<table>
<thead>
<tr>
<th>udepot-syncIO</th>
<th>udepot-aio</th>
<th>udepot-spdk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GET: throughput for qd=128

Throughput (kops)

<table>
<thead>
<tr>
<th>udepot-syncIO</th>
<th>udepot-aio</th>
<th>udepot-spdk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Embedded uDepot: Scalability (20 cores, 24 Flash NVMeS)
Embedded uDepot: Scalability (20 cores, 24 Flash NVMe)

Throughput (Mops/sec)

**GETs**
- **udepot-syncIO**
- **udepot-aio**
- **udepot-spdk**
- **spdk raw**
- **aio raw**

**PUTs**
- **udepot-syncIO**
- **udepot-aio**
- **udepot-spdk**
- **spdk raw**
- **aio raw**

Concurrency (trt:128 × #threads, linux: #threads)

Concurrency (trt:32 × #threads, linux: #threads)
Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)

GETs

PUTs

GET: udepot-syncIO: 1.6 Mops
Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)

GET: udepot-syncIO: 1.6 Mops

PUT: udepot-syncIO: 1.6 Mops

GET: SPDK:
≈ 6.2 Mops (uDepot) vs
≈ 6.9 Mops (ubench)
Evaluation

1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB)
4. uDepot memcache vs DRAM-based memcache implementations
Evaluation

1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB) [see paper]
4. uDepot memcache vs DRAM-based memcache implementations
1. Performance of uDepot index (with and without resize)
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB) [see paper]
4. uDepot memcache vs DRAM-based memcache implementations

**Experiment**

- memaslap benchmark
- default workload: 1KiB objects, 10%/90% PUT/GET
- 2 Optane SSDs, 20 cores, 10GbE
- vs Memcached (expected performance), Memc3 (optimized memcached), Fatcache (traditional SSD impl.)
uDepot memcache

Latency (us) for qd=1

- memc3: 49
- memcached: 51
- udepot-trt-aio: 67
- udepot-trt-spdk: 51
Latency (us) for qd=128
- memc3: 110
- memcached: 126
- uDepot-trt-aio: 139
- uDepot-trt-spdk: 128

Throughput (kiops) for qd=128
- memc3: 1145
- memcached: 1000
- uDepot-trt-aio: 911
- uDepot-trt-spdk: 985
Fast NVMe devices offer an attractive cost-performance tradeoff between DRAM and Flash SSDs.

- They shift the bottleneck from the storage to the network.

uDepot: a KV store that delivers low latency, high throughput.

uDepot memcache has comparable performance with DRAM memcache.

Experimental Cloud service based on uDepot memcache implementation.

Try it out (for free):
https://cloud.ibm.com/catalog/services/data-store-for-memcache
Conclusions

- Fast NVMe devices offer an attractive cost-performance tradeoff between DRAM and Flash SSDs
  - They shift the bottleneck from the storage to the network
- uDepot: a KV store that delivers their performance
  - low latency, high throughput
  - uDepot memcache has comparable performance with DRAM memcache
- Experimental Cloud service based on uDepot memcache implementation
  - try it out (for free): https://cloud.ibm.com/catalog/services/data-store-for-memcache
Conclusions

- Fast NVMe devices offer an attractive cost-performance tradeoff between DRAM and Flash SSDs
  - They shift the bottleneck from the storage to the network
- uDepot: a KV store that delivers their performance
  - low latency, high throughput
  - uDepot memcache has comparable performance with DRAM memcache
- Experimental Cloud service based on uDepot memcache implementation
  - try it out (for free):
Thank you! Questions?