Reaping the performance of fast NVM storage with uDepot

Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas

IBM Research, Zurich

FAST'19

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)

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).

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

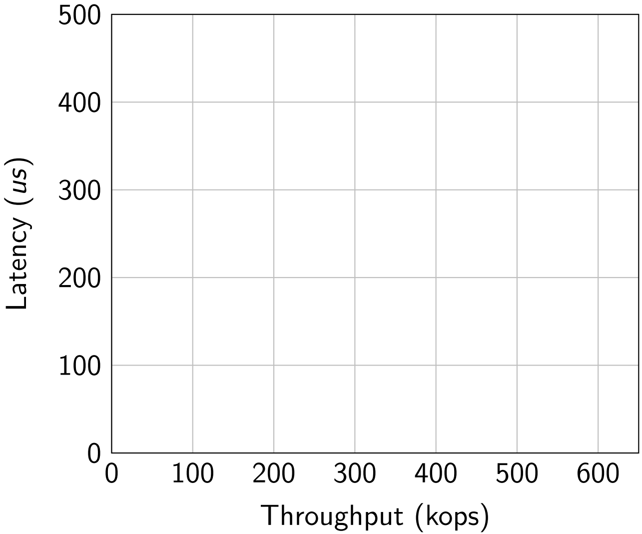
Fast NVM Devices

(FNDs)

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

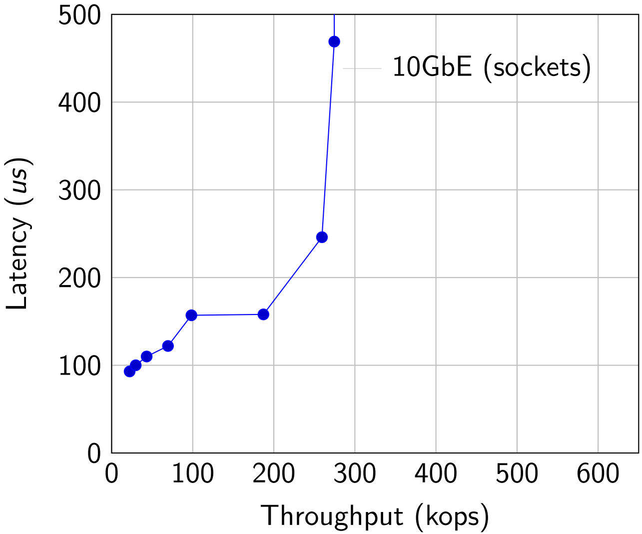
- An order of magnitude better performance than Flash SSDs
- Significantly cheaper than DRAM
  - \$1.25 vs \$10 per GiB (Intel Optane)
  - smaller TCO (number of machines, energy, etc.)

10GiB network vs Flash SSD vs FND SSD



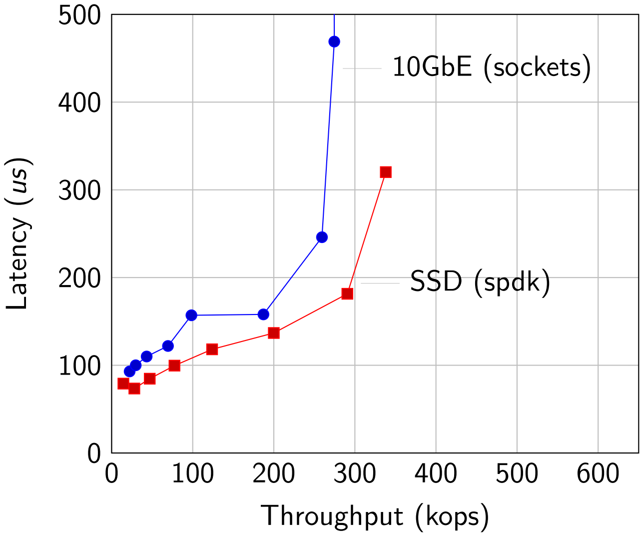
- queue depth (reqs in flight): $1, 2, 4, 8, 16, \ldots$

10GiB network vs Flash SSD vs FND SSD



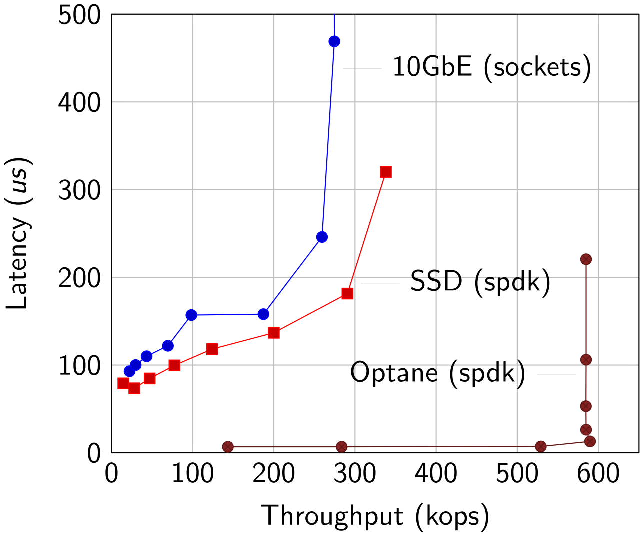
- queue depth (reqs in flight): $1, 2, 4, 8, 16, \ldots$
- *10GbE*: req:1b, res:4KiB (Intel X710, `netperf`)

10GiB network vs Flash SSD vs FND SSD



- queue depth (reqs in flight): $1, 2, 4, 8, 16, \ldots$
- *10GbE*: req:1b, res:4KiB (Intel X710, `netperf`)
- *SSD*: SPDK `perf`: 4KiB RDs (Flash NVMe)

10GiB network vs Flash SSD vs FND SSD



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

→ 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

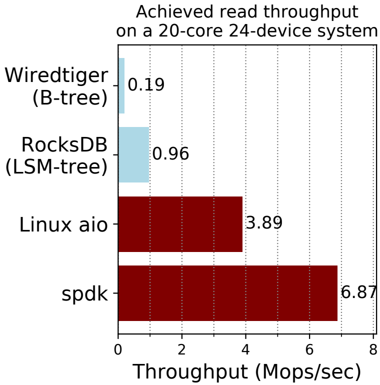
- Built for slower devices
  (e.g., use synchronous IO)
- Data structures with inherent IO amplification
  (LSM- or B-trees)
- Cache data in DRAM, limiting scalability
- Rich feature set (e.g., transactions, snapshots)

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)



uDepot: a KV store for FNDs

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.

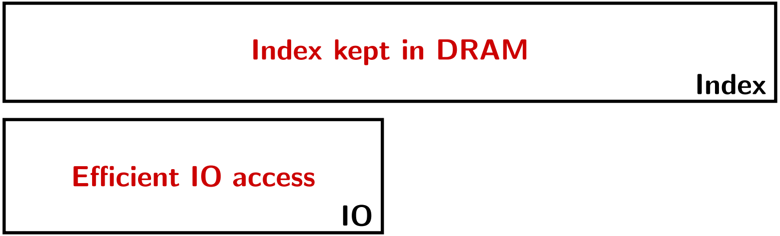
uDepot architecture

uDepot architecture

Index kept in DRAM

Index

uDepot architecture



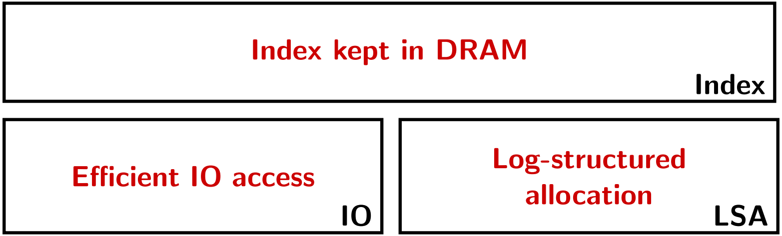
Index kept in DRAM

Index

Efficient IO access

IO

uDepot architecture



Index kept in DRAM

Index

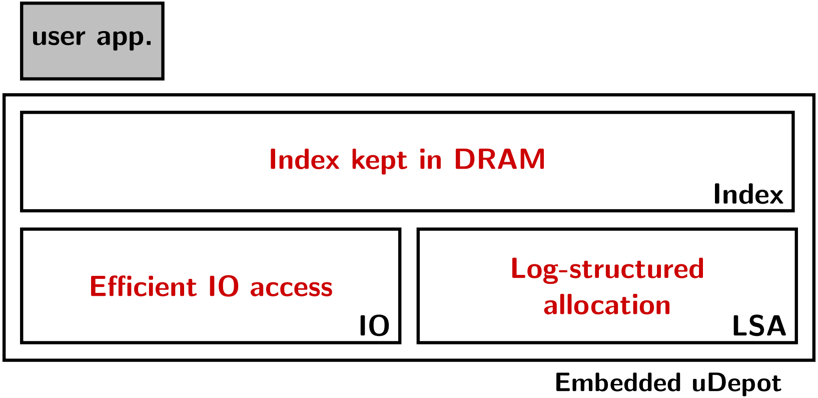
Efficient IO access

IO

Log-structured allocation

LSA

uDepot architecture



user app.

Index kept in DRAM

Index

Efficient IO access

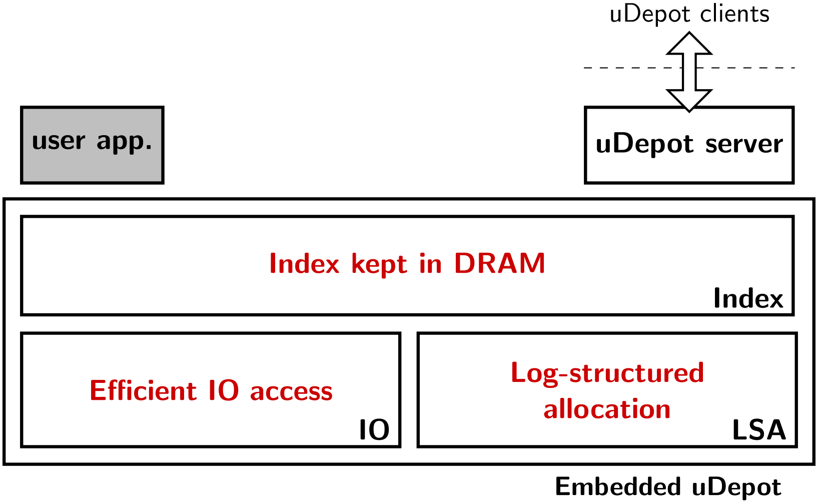
IO

Log-structured allocation

LSA

Embedded uDepot

uDepot architecture



uDepot clients

user app.

uDepot server

Index kept in DRAM

Index

Efficient IO access

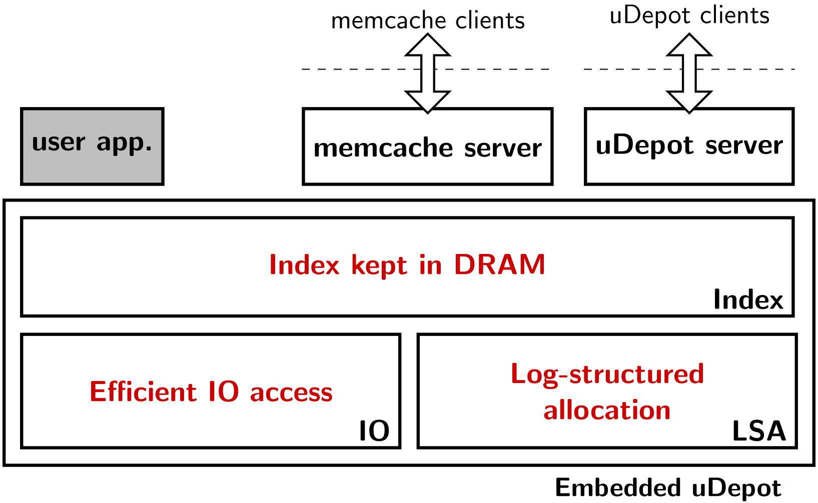
IO

Log-structured allocation

LSA

Embedded uDepot

uDepot architecture



memcache clients

uDepot clients

user app.

memcache server

uDepot server

Index kept in DRAM

Index

Efficient IO access

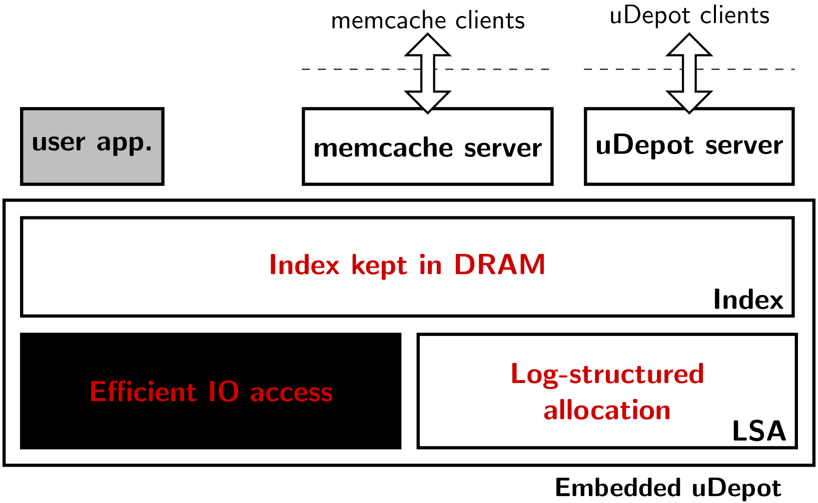
IO

Log-structured allocation

LSA

Embedded uDepot

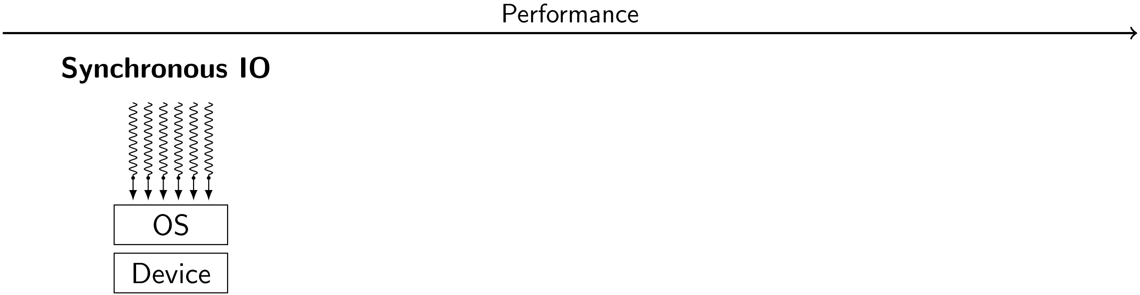
uDepot architecture



IO Facilities

Performance

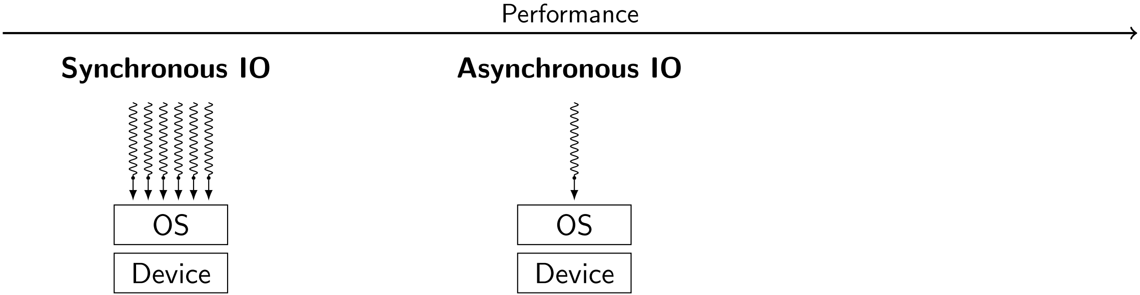
IO Facilities



Synchronous IO

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

IO Facilities



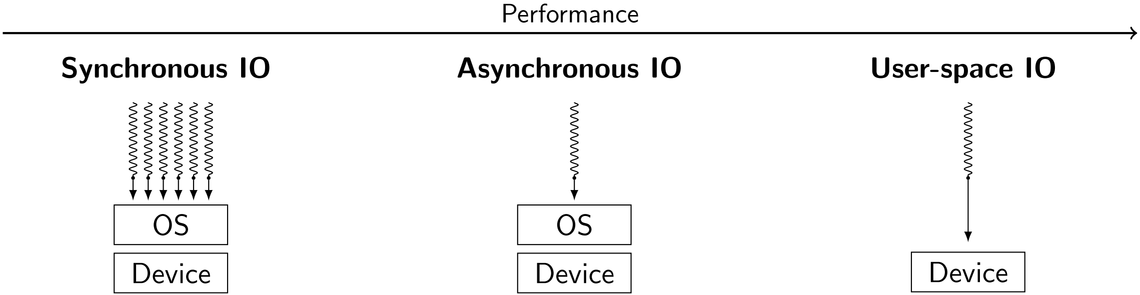
Synchronous IO

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

Asynchronous IO

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

IO Facilities



Synchronous IO

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

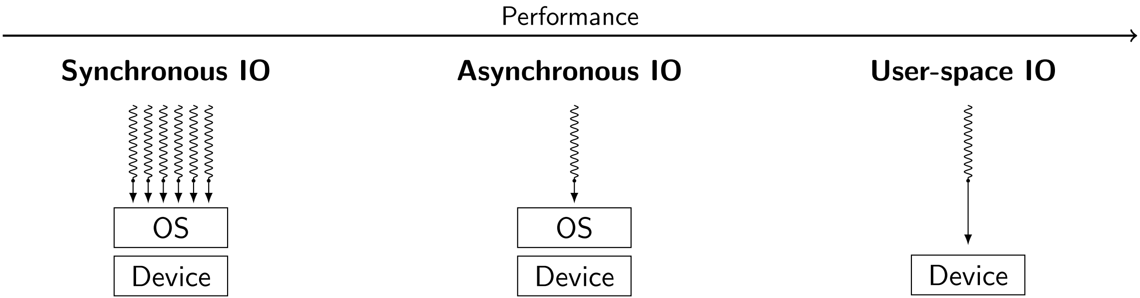
Asynchronous IO

- 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



Synchronous IO

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

Asynchronous IO

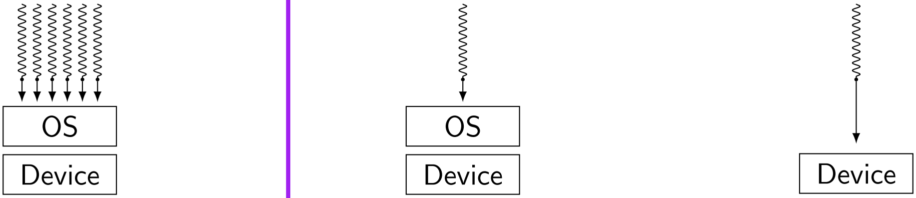
- issue IO requests
- receive IO completions
- e.g., Linux AIO
- uDepot aio backend

User-space IO

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

IO Facilities

Performance →



Synchronous IO

- one thread per request
- syncronous (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

TRT: a run-time system for async IO

TRT: A task run-time system for asynchronous IO

Goals

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

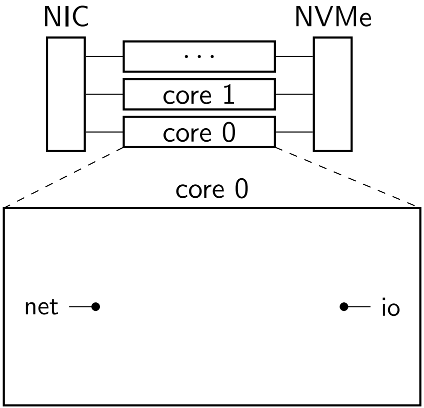
TRT: A task run-time system for asynchronous IO

Goals

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

TRT highlights

- avoid cross-core communication



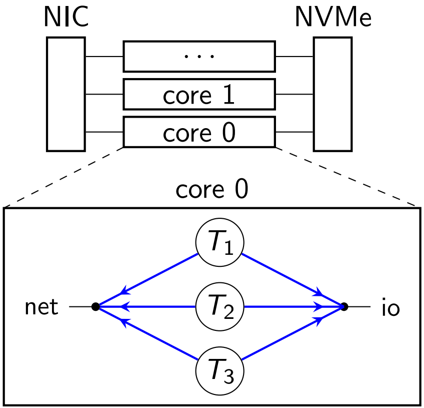
TRT: A task run-time system for asynchronous IO

Goals

- progammer-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)



Tasks (T) issue IO requests →

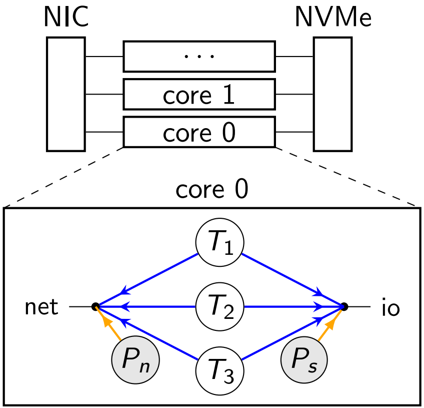
TRT: A task run-time system for asynchronous IO

Goals

- progammer-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 →

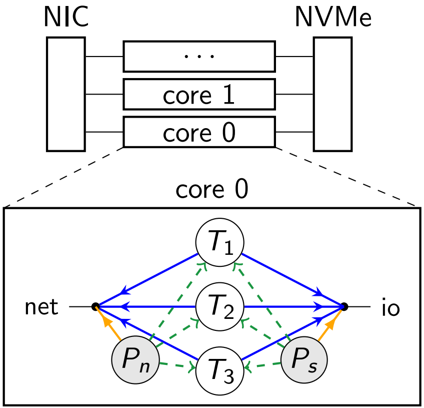
TRT: A task run-time system for asynchronous IO

Goals

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

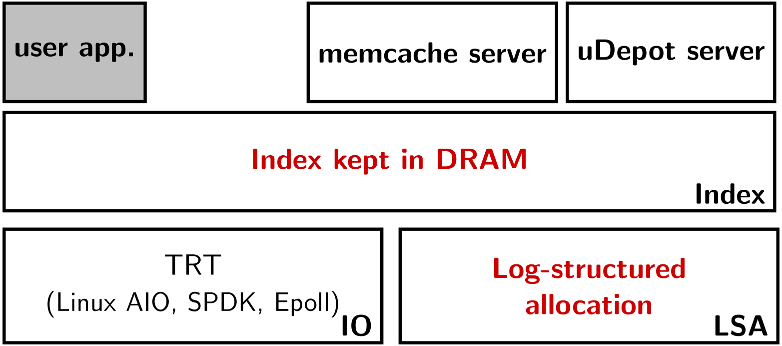
TRT highlights

- 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 - - ›

uDepot architecture



user app.

memcache server

uDepot server

Index kept in DRAM

Index

TRT

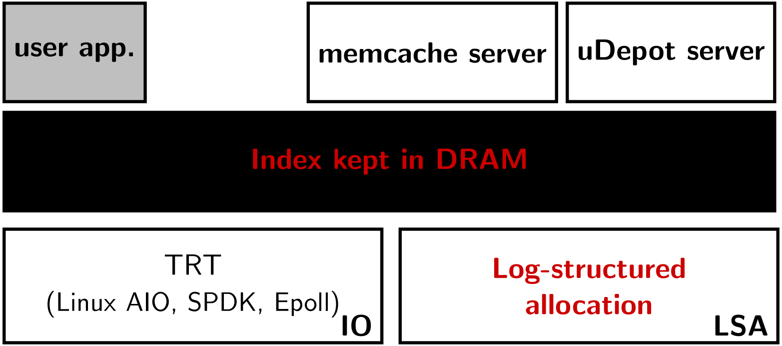
(Linux AIO, SPDK, Epoll)

IO

Log-structured allocation

LSA

uDepot architecture



user app.

memcache server

uDepot server

Index kept in DRAM

TRT

(Linux AIO, SPDK, Epoll)

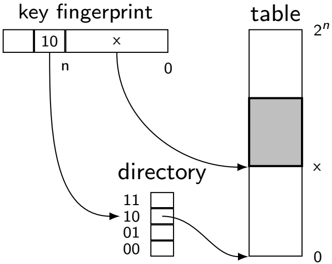
IO

Log-structured allocation

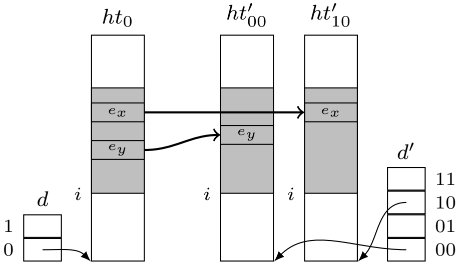
LSA

uDepot index

- 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



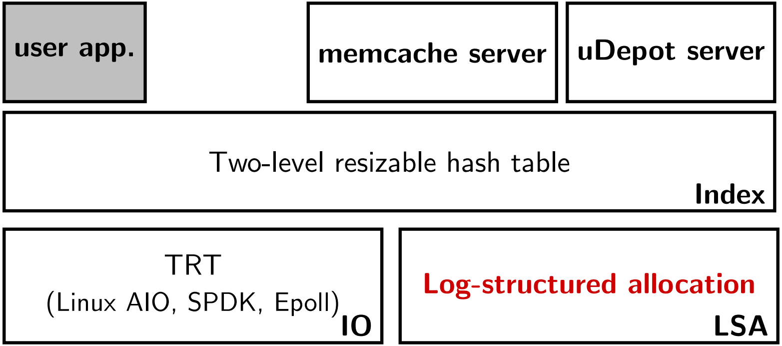
Operations:

- 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

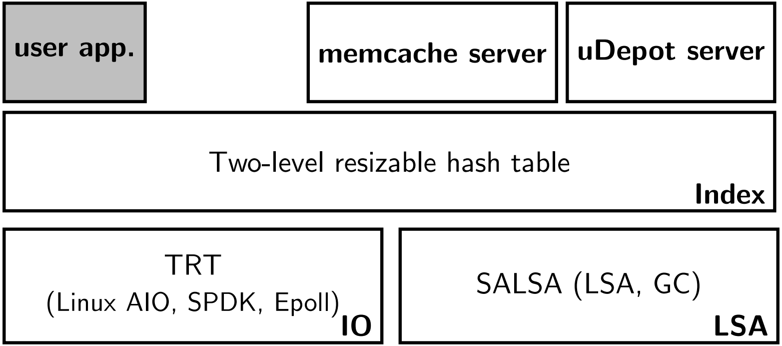
TRT
(Linux AIO, SPDK, Epoll)

IO

Log-structured allocation

LSA

uDepot architecture



user app.

memcache server

uDepot server

Two-level resizable hash table

Index

TRT
(Linux AIO, SPDK, Epoll)

IO

SALSA (LSA, GC)

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, ScyllaDB)
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

uDepot index latency

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

uDepot index latency

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

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, ScyllaDB)
4. uDepot memcache vs DRAM-based memcache implementations

Evaluation

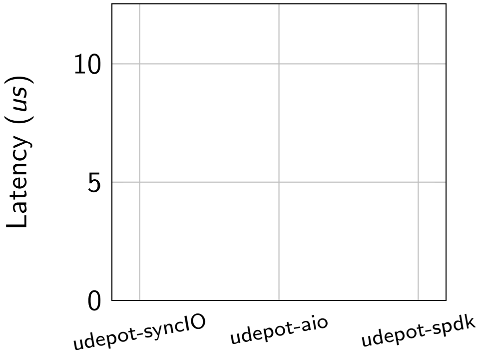
1. ~~Performance of uDepot index (with and without resize)~~
2. Embedded uDepot performance vs device performance
3. uDepot server vs other NVMe stores (Aerospike, ScylaDB)
4. uDepot memcache vs DRAM-based memcache implementations

Experiment

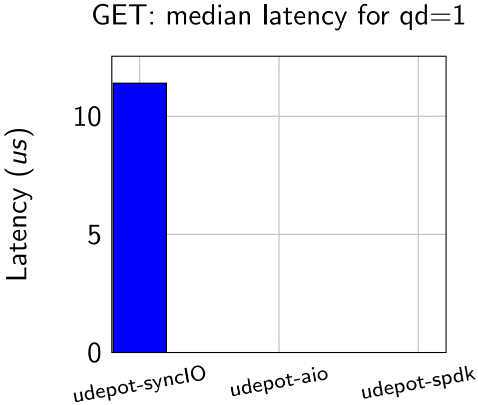
- uDepot ubench: perform 10M uDepot `PUTs` and `GETs`
    - multiple backends (how different backends behave)
- vs. dev ubench: `fio` and SPDK `perf`
- same workload: 4KiB

Embedded uDepot: Efficiency (1 core / 1 Optane)

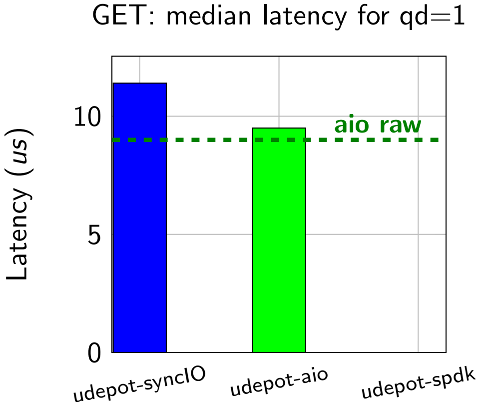
GET: median latency for qd=1



Embedded uDepot: Efficiency (1 core / 1 Optane)

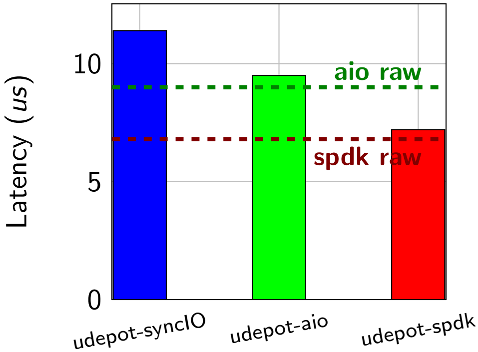


Embedded uDepot: Efficiency (1 core / 1 Optane)

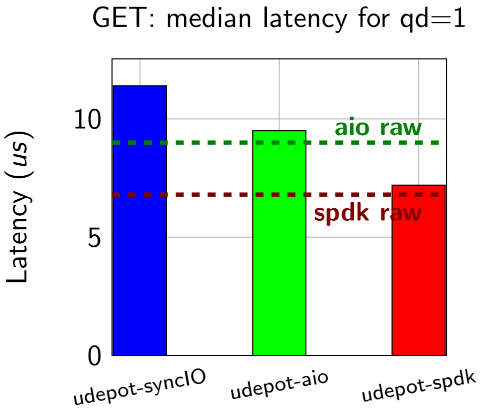


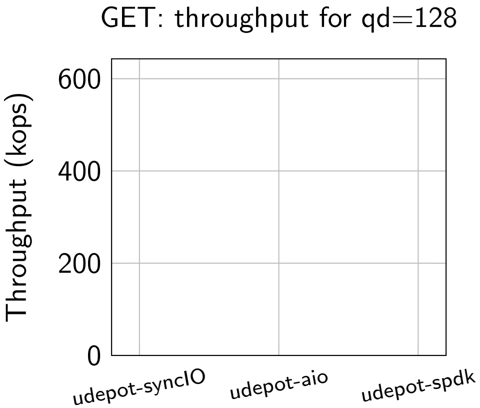
Embedded uDepot: Efficiency (1 core / 1 Optane)

GET: median latency for qd=1

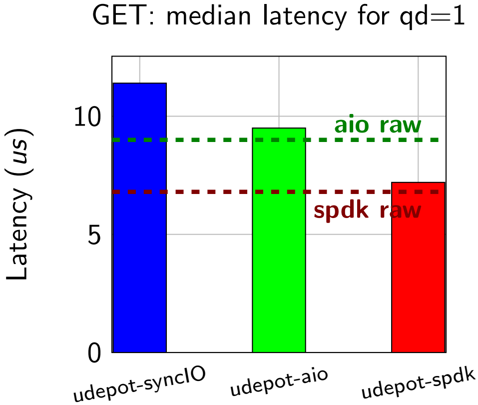


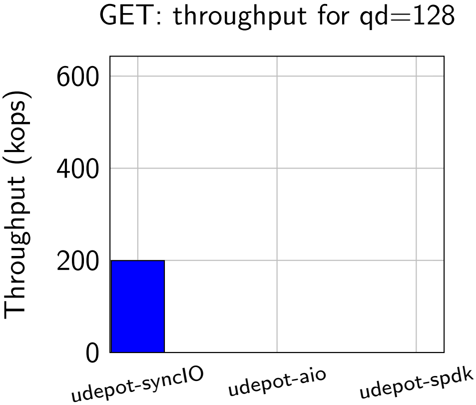
Embedded uDepot: Efficiency (1 core / 1 Optane)



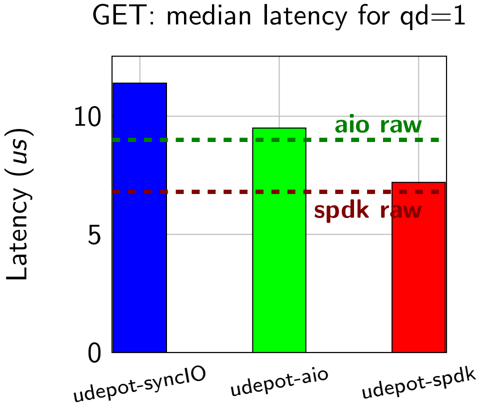


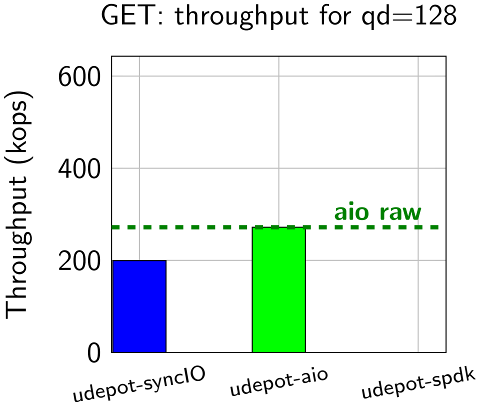
Embedded uDepot: Efficiency (1 core / 1 Optane)



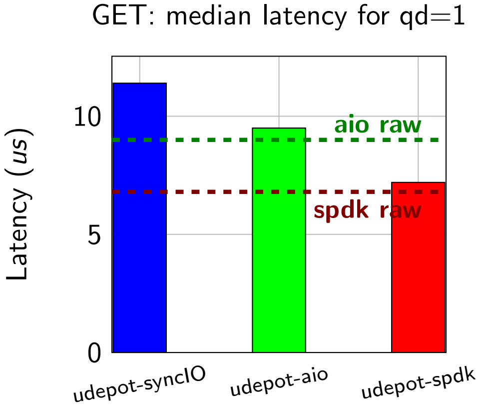


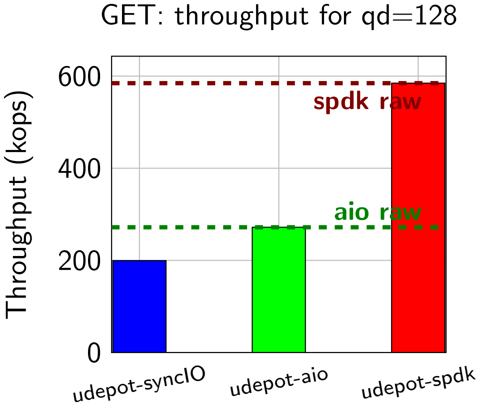
Embedded uDepot: Efficiency (1 core / 1 Optane)





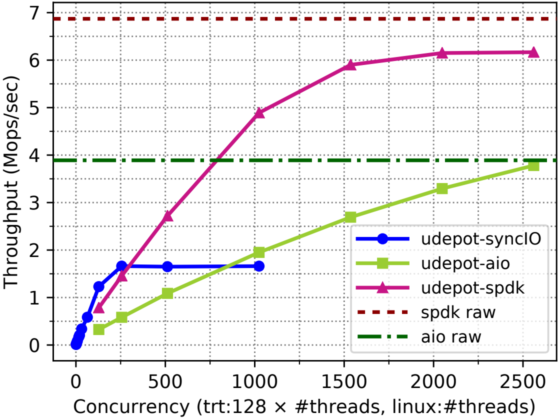
Embedded uDepot: Efficiency (1 core / 1 Optane)



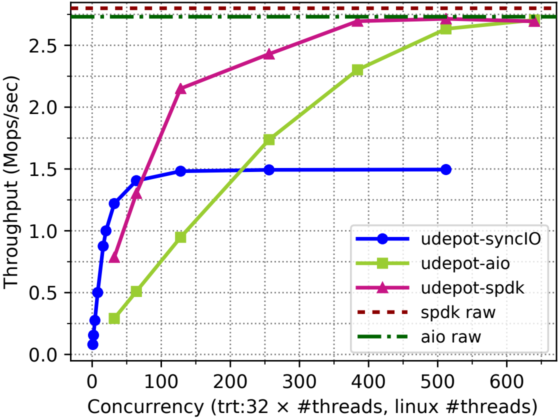


Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)

Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)

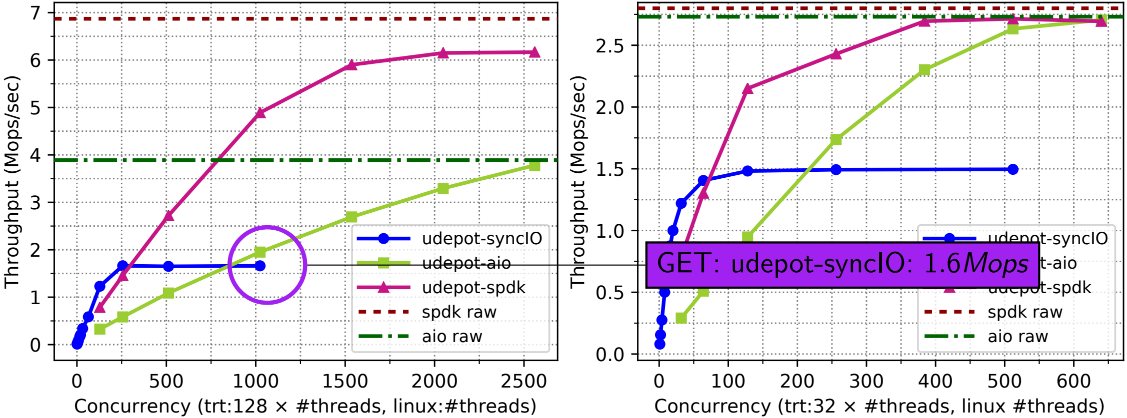


GETs



PUTs

Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)

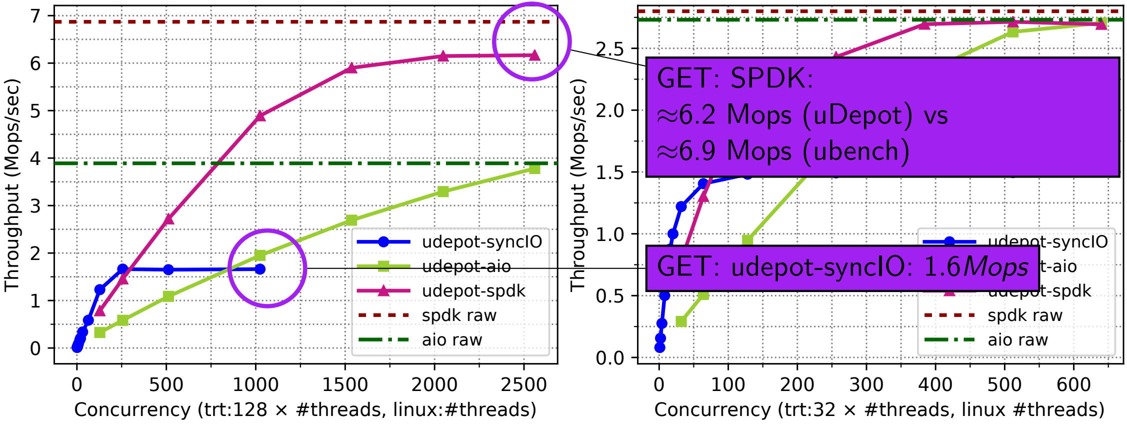


GETs

PUTs

GET: udepot-syncIO: 1.6$Mops$

Embedded uDepot: Scalability (20 cores, 24 Flash NVMes)



GETs

PUTs

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

GET: udepot-syncIO: 1.6*Mops*

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, ScyllaDB)
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

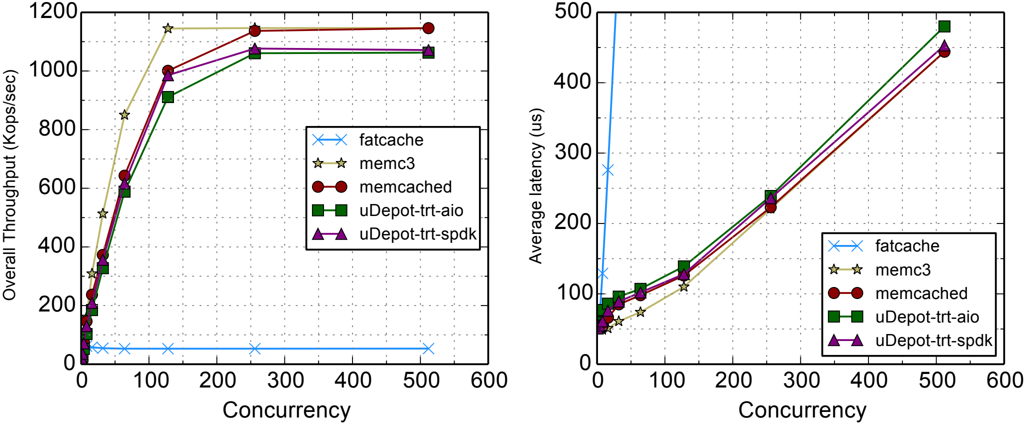
Evaluation

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

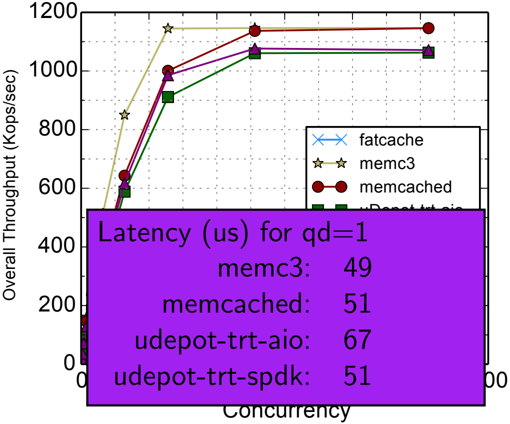
Experiment

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

uDepot memcache

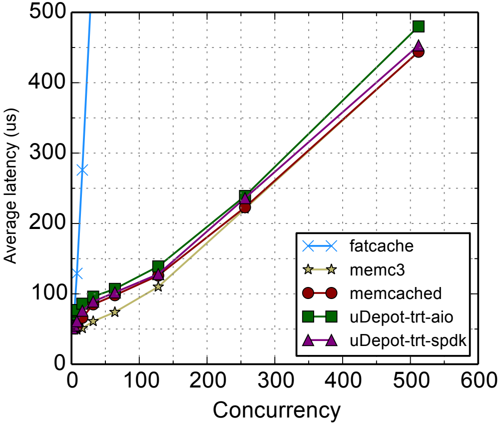


uDepot memcache

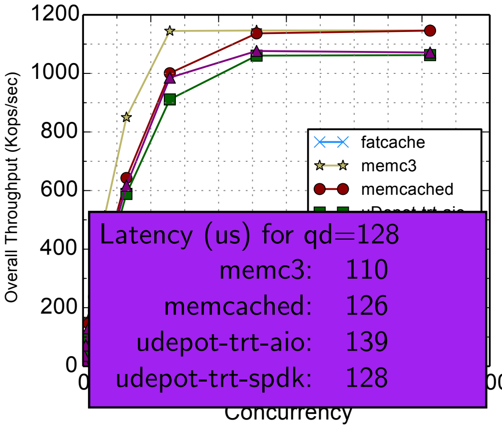


Latency (us) for qd=1

memc3:	49
memcached:	51
udepot-trt-aio:	67
udepot-trt-spdk:	51

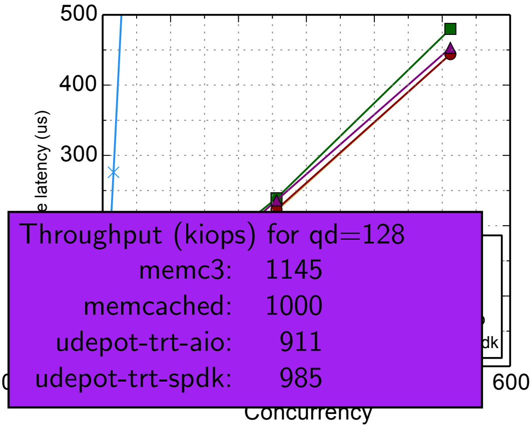


uDepot memcache



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

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

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

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`

Thank you! Questions?