uDepot: Key-Value Store for Fast NVM Storage

Kornilios Kourtis*, Nikolas Ioannou*, Radu Stoica*,
Joshua Mintz+, Haris Pozidis*

*IBM Zurich Research +IBM Cloud

Outline

uDepot: A KV store for Fast NVM Drives

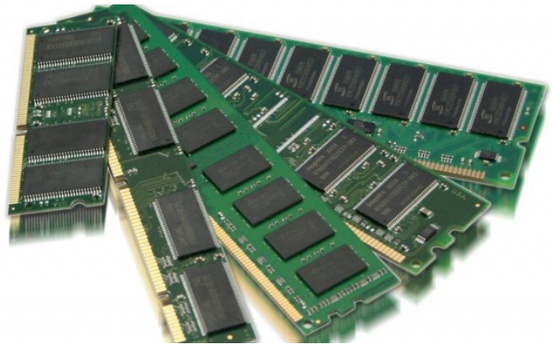
- Motivation
- Architecture
- Evaluation

Cloud caching service

- Experimental service on the IBM Cloud built using uDepot
- memcache API compatible

DRAM Key-Value stores

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



- DRAM is not getting cheaper or denser

- DRAM performance is underutilized
  - Commodity networks (e.g., 10GbE) are the bottleneck
  - High-performance DRAM KV stores use: RDMA (RamCloud, FaRM), Direct NIC access(MICA), programmable NICs (KV-Direct)

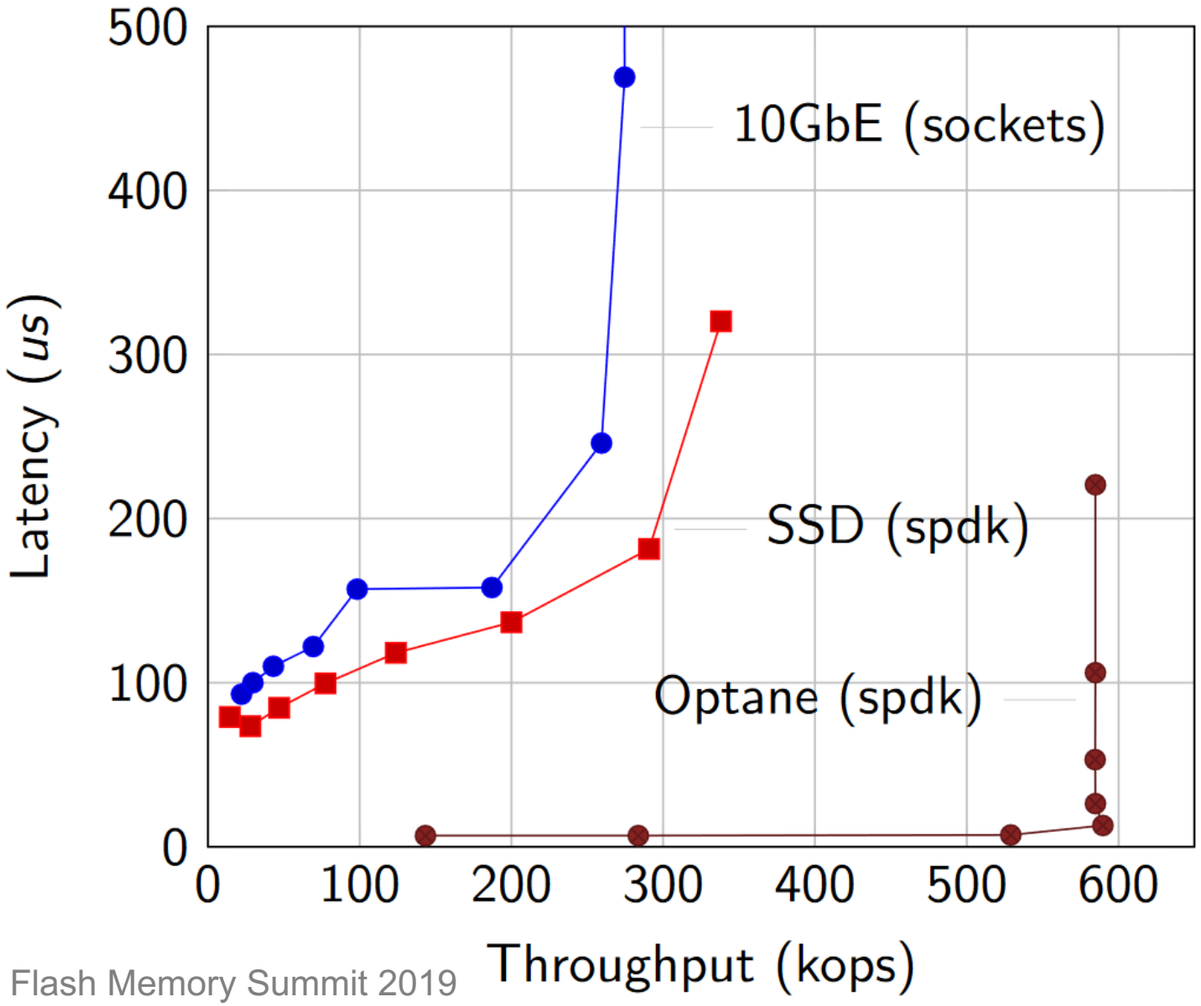
Fast NVM Devices (FNDs)

- New classes of NVM-based SSDs:
  - Intel Optane (3DXP)
  - Samsung Z-SSD (Z-NAND)
  - Others

- New level in the memory hierarchy:
  - Order of magnitude better performance than Flash SSDs
  - Significantly cheaper than DRAM
  - Lower TCO due to increased density (number of machines, energy, etc.)

Technology	Cost ($ / GB)
DRAM	$9.8
Optane NVMe	$1.25
Flash NVMe	$0.4

10 GbE vs. NVMe SSD vs. Optane



Hardware:

- 10GbE: Intel X710
- SSD: Intel P3600
- FND: Intel Optane

Workloads:

- 10GbE: Netperf (request 1B, result 4KiB)
- SSD, FND: SPDK perf (4KB reads)

Methodology:

- Each datapoint shows a given queue depth (requests in flight 1, 2, 4, … 128)

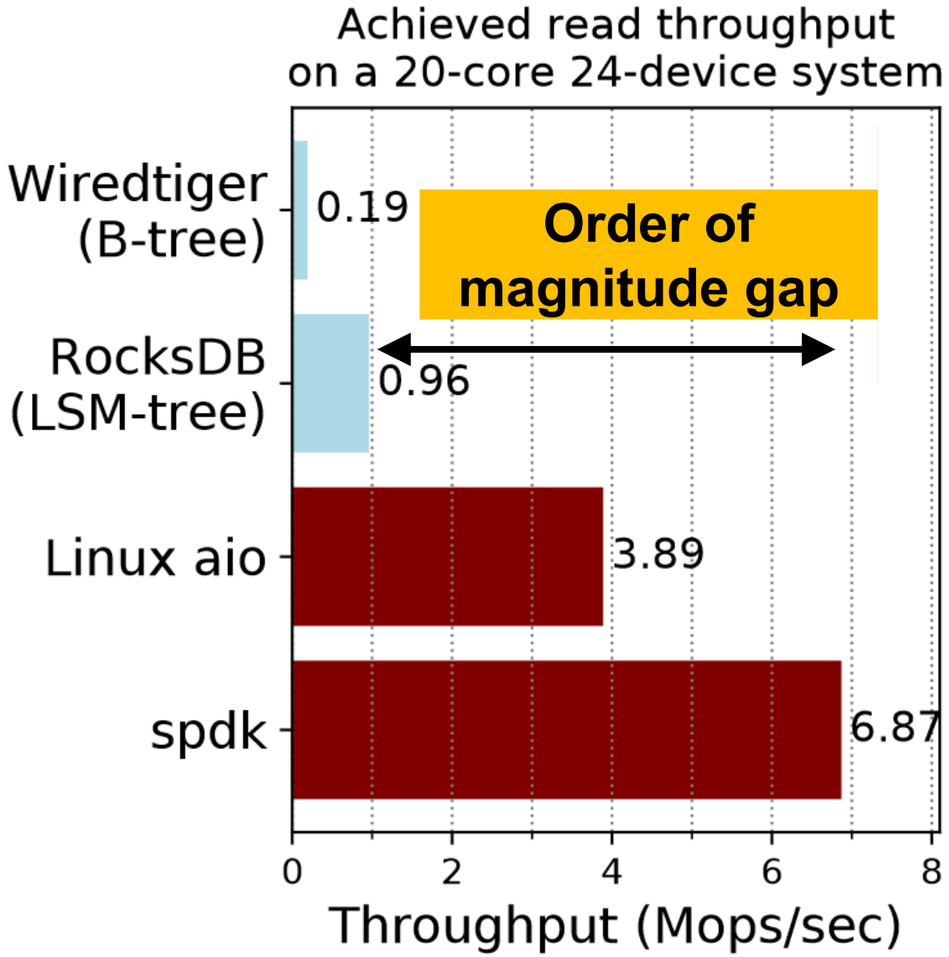
FNDs in existing KV stores

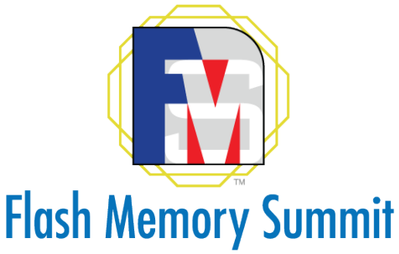
Goals:

- Reduced cost
- Equivalent performance to DRAM KV stores (at least, under commodity networks)

Existing KV stores cannot deliver FND performance:

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



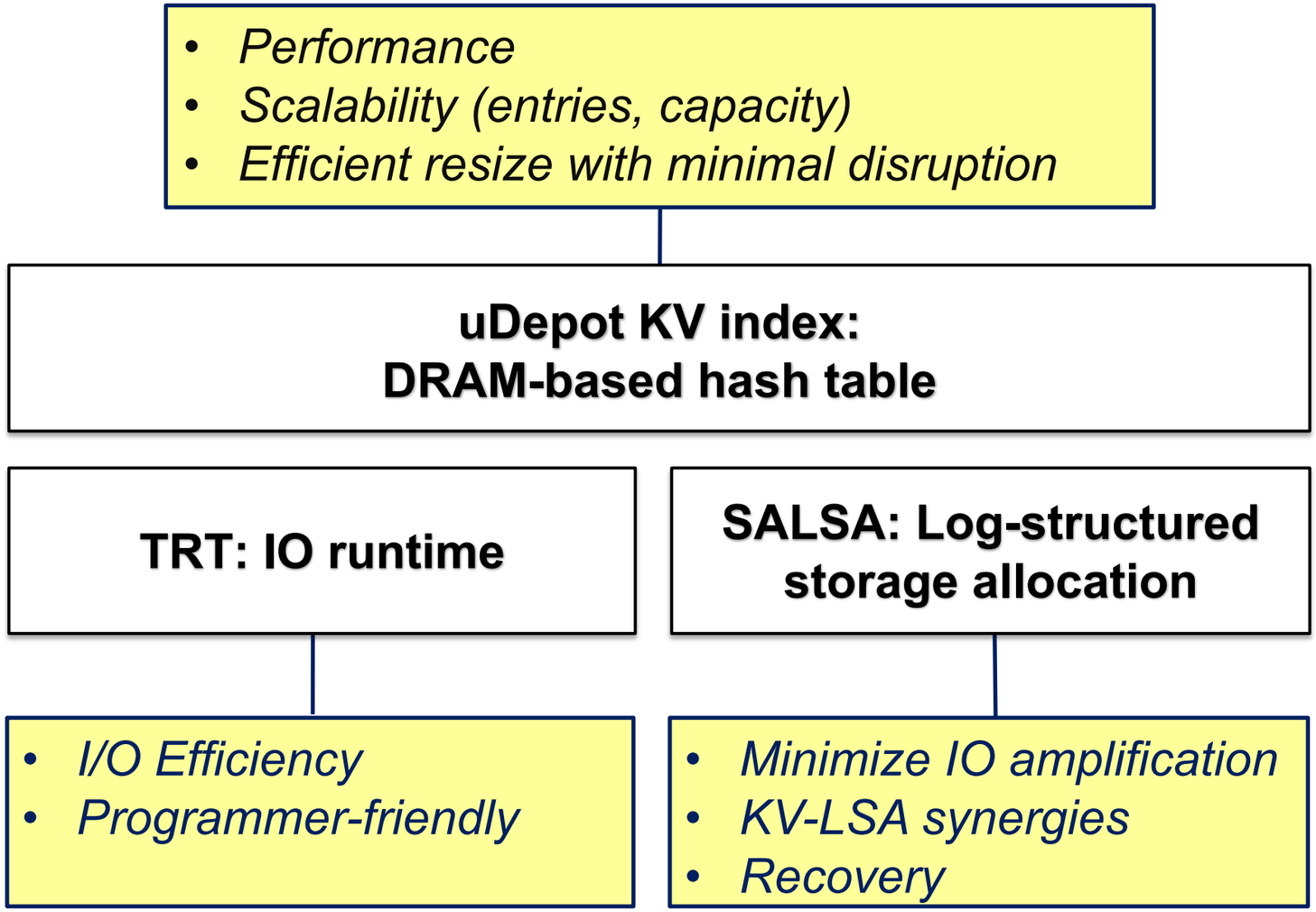


uDepot: a KV store for FNDs

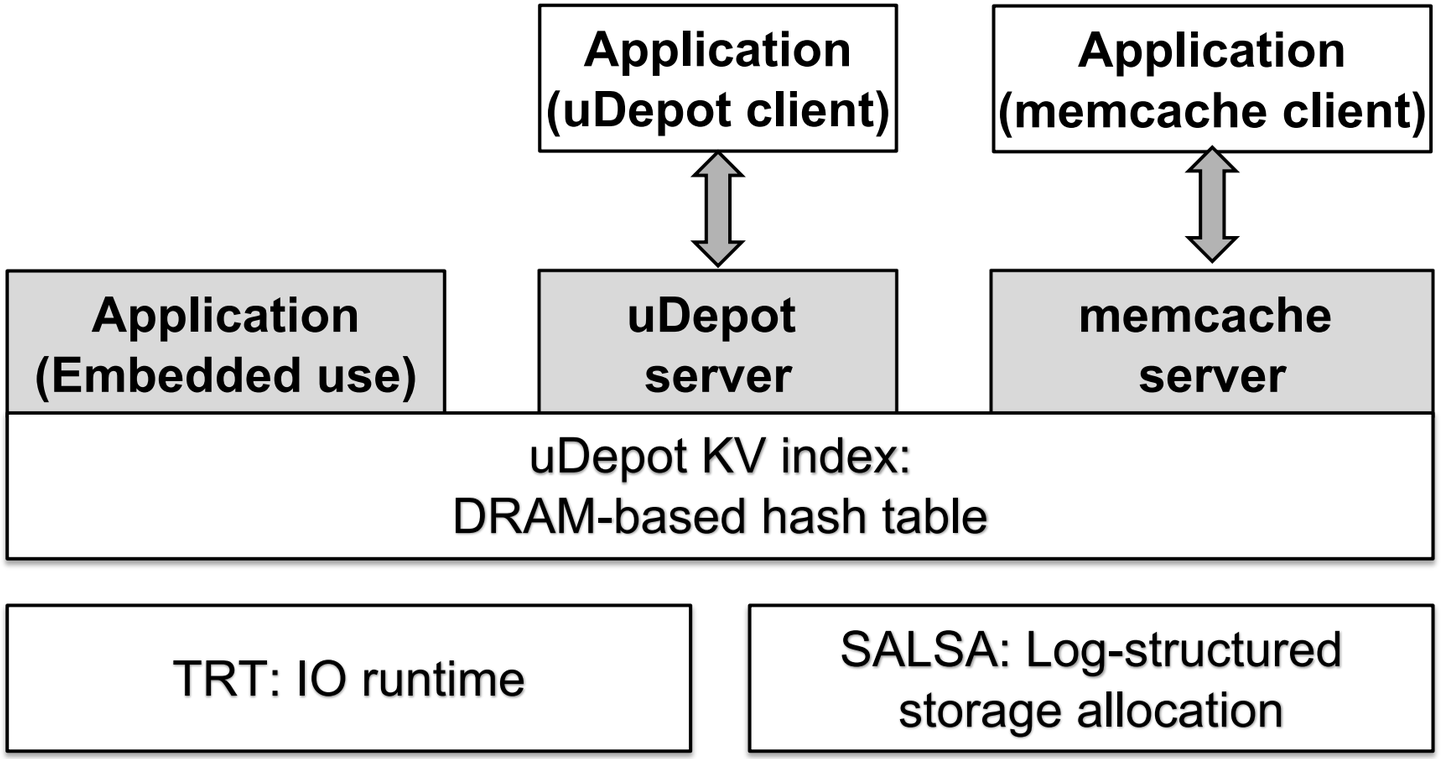
Delivers the performance of FNDs to the application:

- Bottom-up approach
  - Basic interface: GET, PUT, DEL on variable-sized keys and values
- Minimizes IO amplification
- CPU efficiency
- Designed to scale (cores, devices, capacity)

uDepot core components



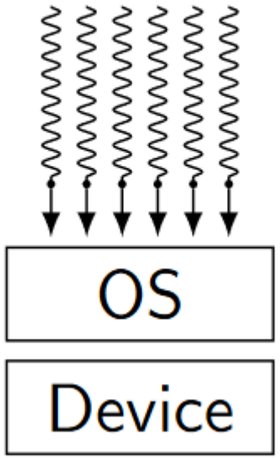
uDepot APIs


Application
(uDepot client)
Application
(memcache client)
Application
(Embedded use)
uDepot
server
memcache
server
uDepot KV index:
DRAM-based hash table
TRT: IO runtime
SALSA: Log-structured
storage allocation

uDepot IO facilities

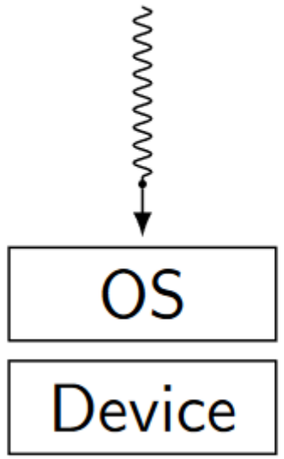
Performance

Synchronous IO



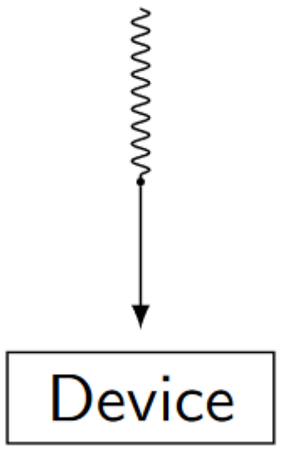
- One thread per request
- Synchronous (blocking) syscalls (e.g., pread)

Asynchronous IO



- Threads issue IO requests and receive IO completions (e.g., Linux AIO)

User-space IO

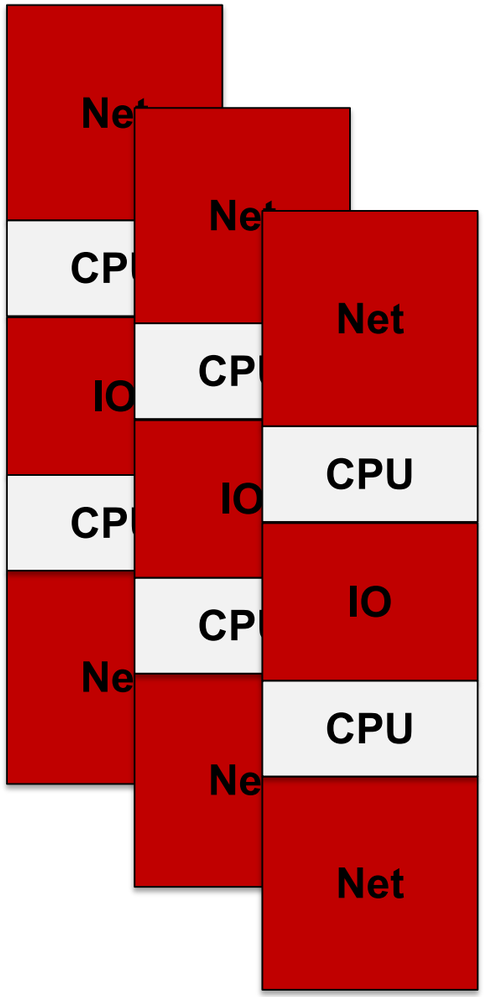


- Directly access the device (OS-bypass)
- Polling instead of interrupts (e.g., SPDK)

TRT: a run-time system for async I/O

uDepot supports backends for all these IO facilities

How to efficiently exploit IO parallelism?




```
while (true) {
   recv(request);
   if (request.is_put())
        … write(request.data); …
   else
        … read(reply.data); …
   send(reply);
}
```

Programming/execution models:

	Simplicity	Efficiency
1. Sync I/O: Use one OS thread per client	✔	✗
2. Async I/O: Callbacks (“stack ripping hell”)	✗	✔

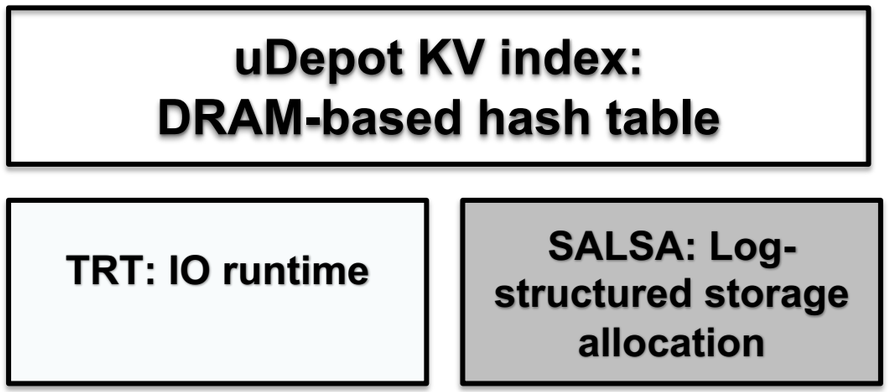
Task Run-Time (TRT) for fast IO

- Provides synchronous-like interface when performing asynchronous IO
- Task-based
  - Each task has its own stack
  - User-space context switching
  - Collaborative scheduling
- Task low-level interface
  - Spawn tasks
  - wait() on events, notify() tasks
- I/O backends (e.g., epoll, aio, spdk)
  - Implement polling via poller tasks

```
while (true) {
  trt::net::recv(request);
  if (request.is_put())
    …trt::spdk::write(request.data);…
  else
    …trt::spdk::read(reply.data);…
  trt::net::send(reply);
}
```

Simplicity ✔ **Efficiency** ✔

uDepot design



SALSA*: log-structured storage

Why log-structured storage (LSA):

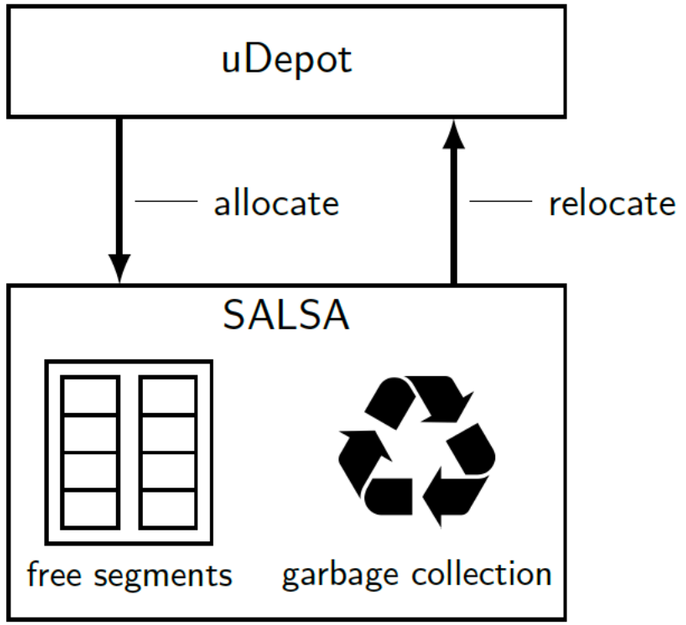
- Minimize write IO amplification & exploit random access
- Out-of-place updates allow recovery and avoid write-ahead logging

Why SALSA:

- Efficient storage allocation and GC policies

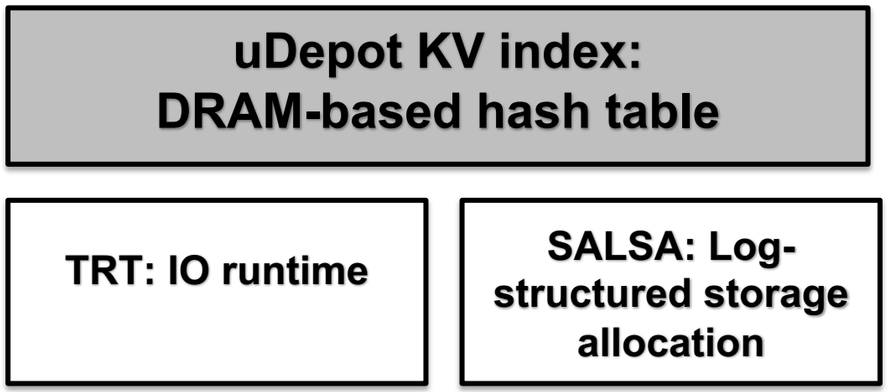
LSA-KV synergy:

- A single logical-to-physical mapping for the data
  (the uDepot hash table is both the KV and LSA index)
- GC performed at the application (uDepot) level
- For caching applications (e.g., Memcache), we codesign GC & cache eviction



*Ioannou et al. "Elevating commodity storage with the SALSA host translation layer." MASCOTS 2018.

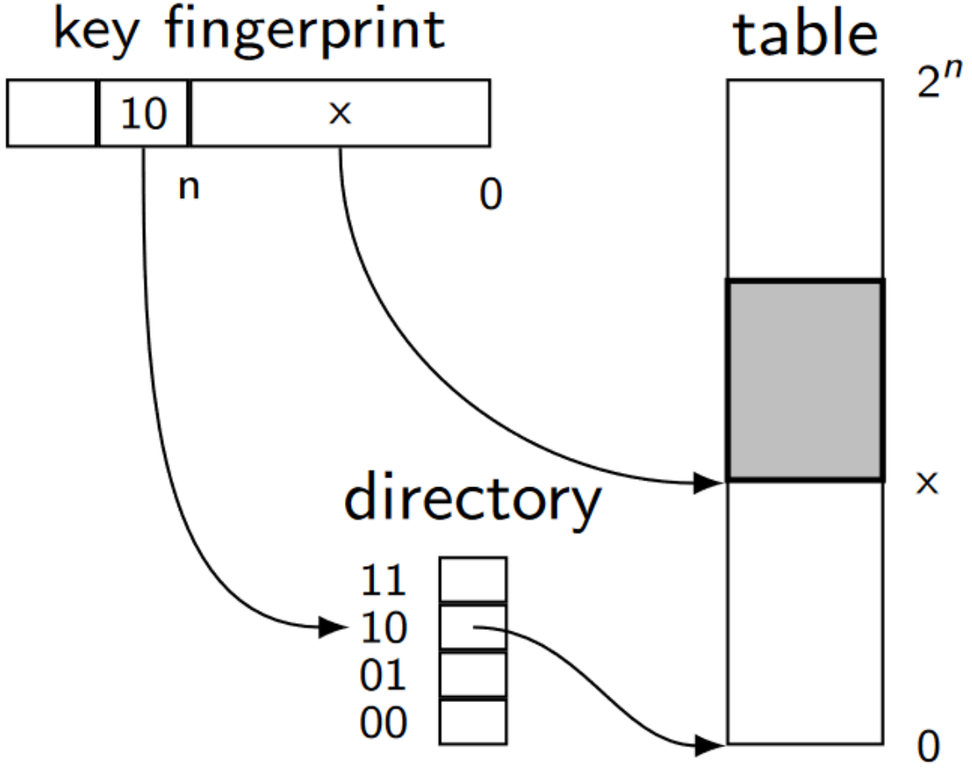
uDepot design



uDepot index data-structure

Features:

- Support for large datasets
  - Two-level structure
  - 8 byte hash entry
  - Resize with minimal disruption
- High-performance:
  - Hopscotch hash table
- Efficient I/O
  - Maintain KV size in the entry



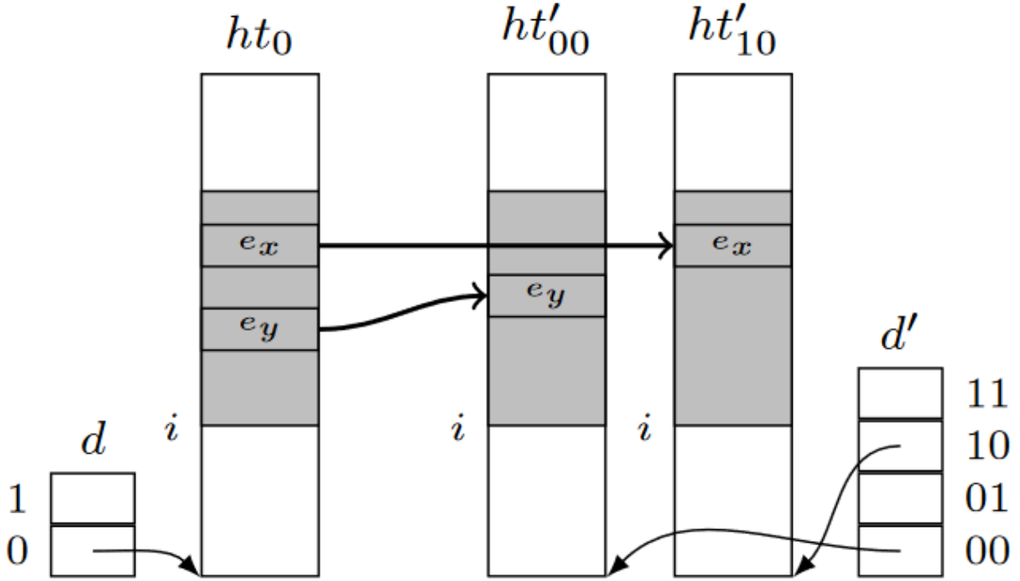
Growing the uDepot index

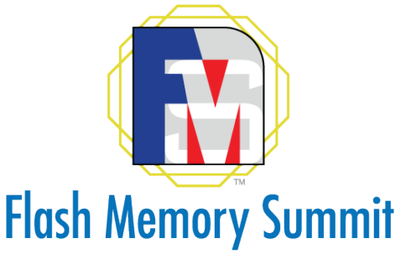
Operation sequence:

1. Double the size of the directory
2. 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 (important to bound tail latency)





uDepot evaluation

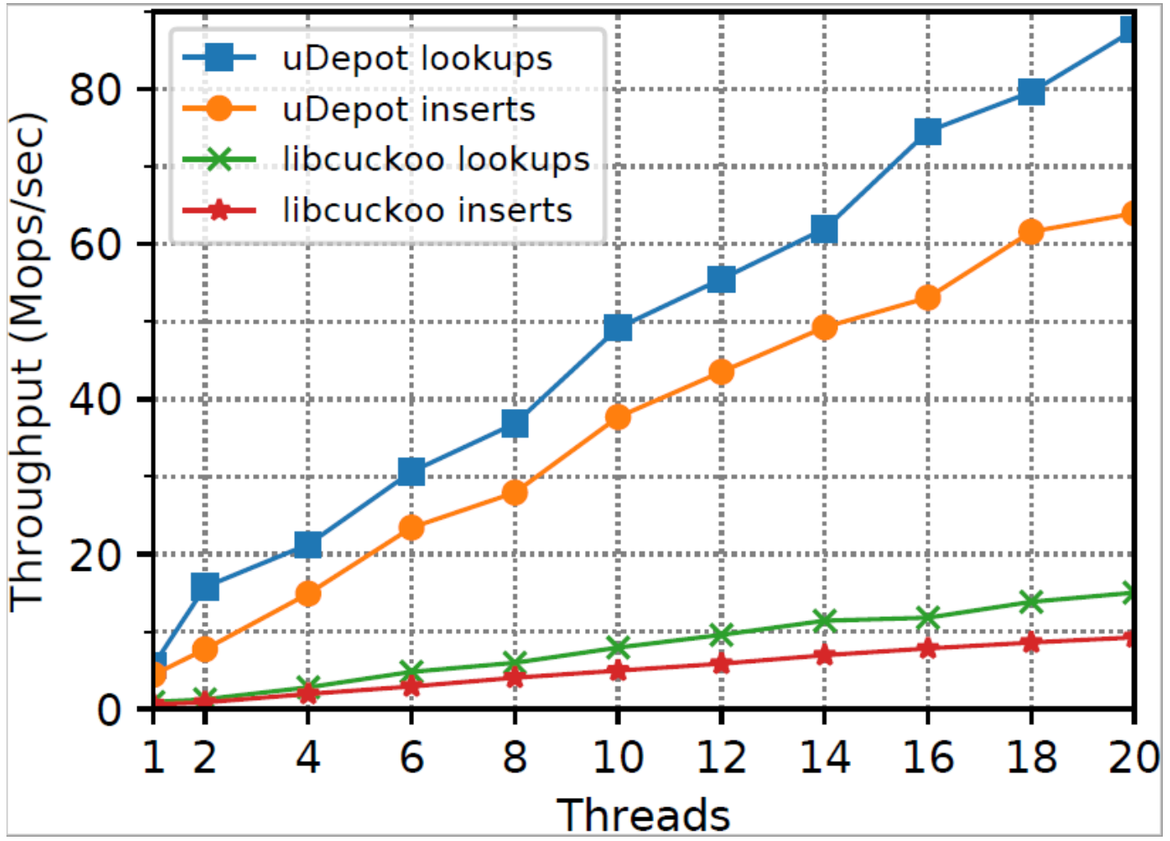
1. How does the uDepot index perform?
   - Compare uDepot index throughput/latency vs. existing hash tables
2. Does uDepot deliver device performance?
   - Compare uDepot throughput/latency vs. device performance
3. Can uDepot replace DRAM-based KV stores?
   - uDepot Memcache API vs. memcache systems (DRAM-based)

Additional details – check our paper [FAST '19] or ask me about:

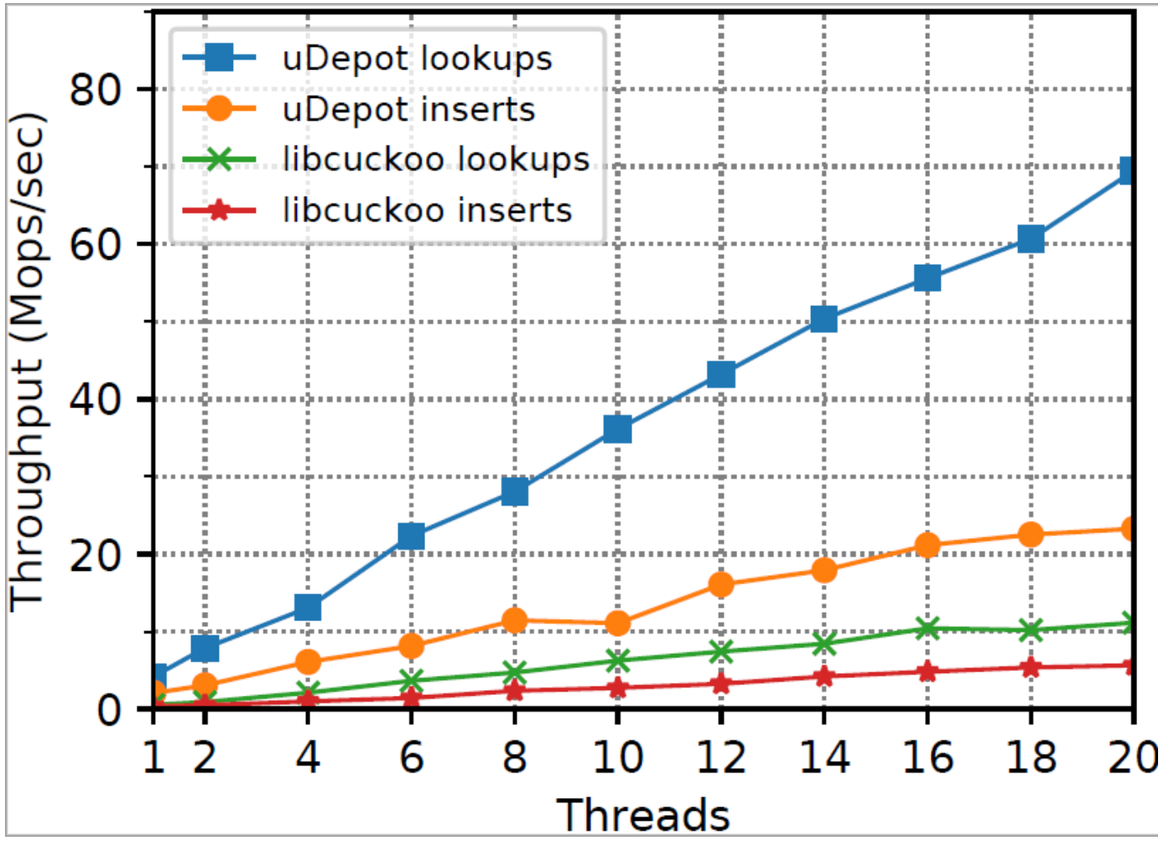
- *uDepot vs. existing SSD-optimized NoSQL stores*

uDepot index throughput

No resize (50M ops)



4 grows (1B ops)



6x more lookups, 7x more inserts

uDepot index latency

	Lookup latency (µs)		Insert latency (µs)	
Percentile	**50M**	**1B**	**50M**	**1B**
50%	0.2	0.3	0.2	0.4
99%	1.1	1.2	0.6	1.0
99.9%	1.9	2.0	1.6	9.2
99.99%	11.0	8.9	7.5	1168.0

Resizing-induced latency: ~1ms at 99.99%

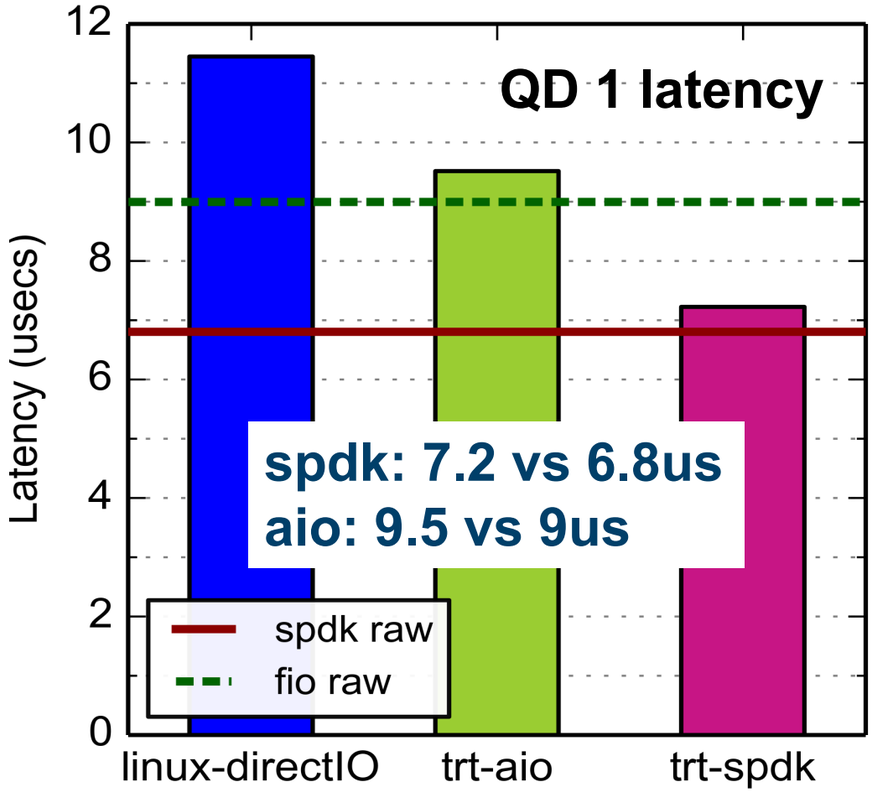
Does uDepot deliver device performance?

Experiment:

- uDepot μbench: 10M 4KB PUTs & GETs
- Exercise all uDepot IO backends
- Compare to the maximum raw device performance
  - *fio (async IO)* and *SPDK perf* running the same IO workload

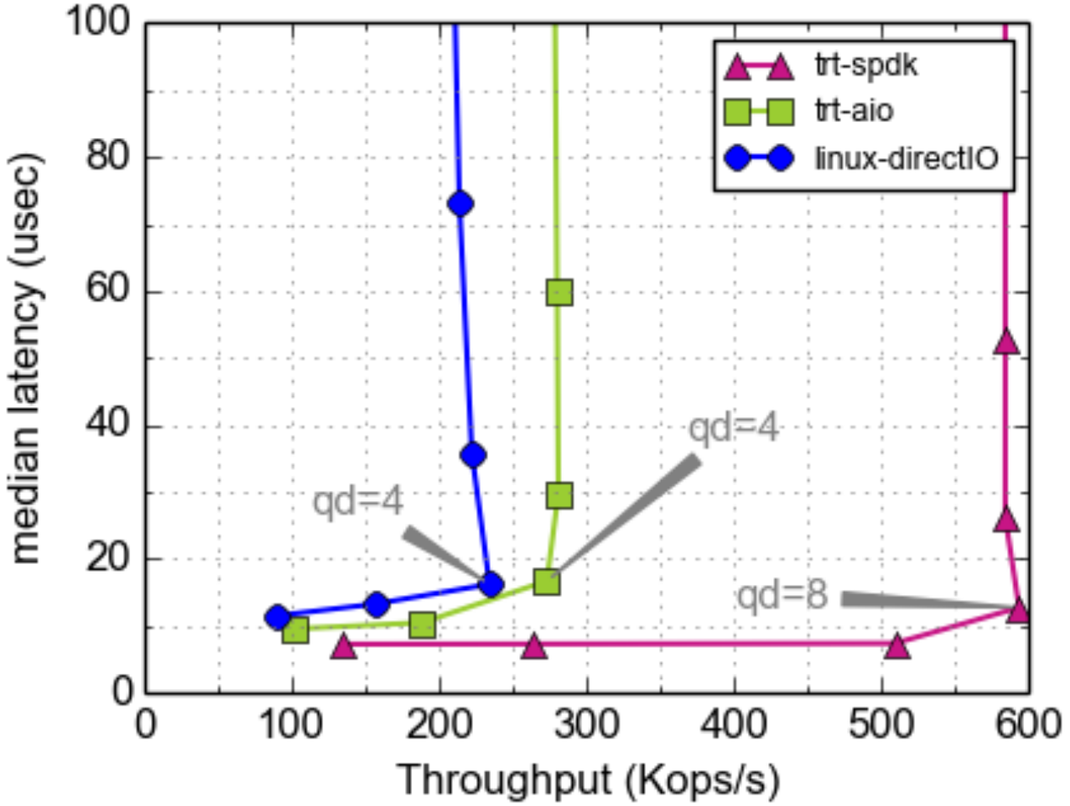
uDepot efficiency

Hardware: 1 core, 1 Optane,



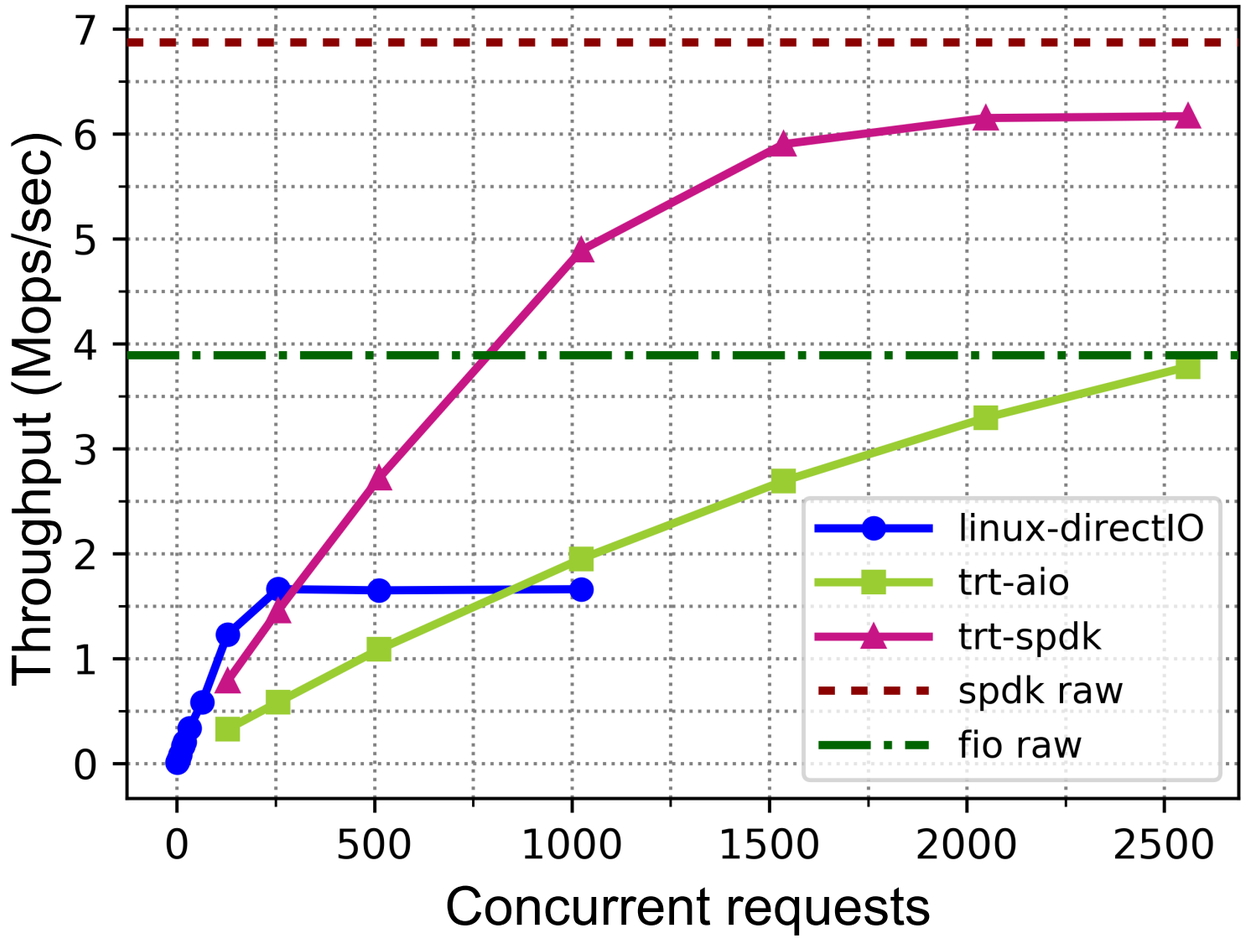
uDepot adds <0.5us latency to the data path

Workload: 10M 4KB PUTs & GETs



Stable latency at high throughput

uDepot scalability



Hardware: 20 cores, 24 NVMe SSDs

uDepot vs. raw device performance:

- AIO: 3.8 vs 3.9 Mops/s
- SPDK: 6.2 vs 6.9 Mops/s

uDepot delivers 6M+ random GETs (24 GB/s) on a single machine!

Can uDepot replace DRAM-based KV stores?

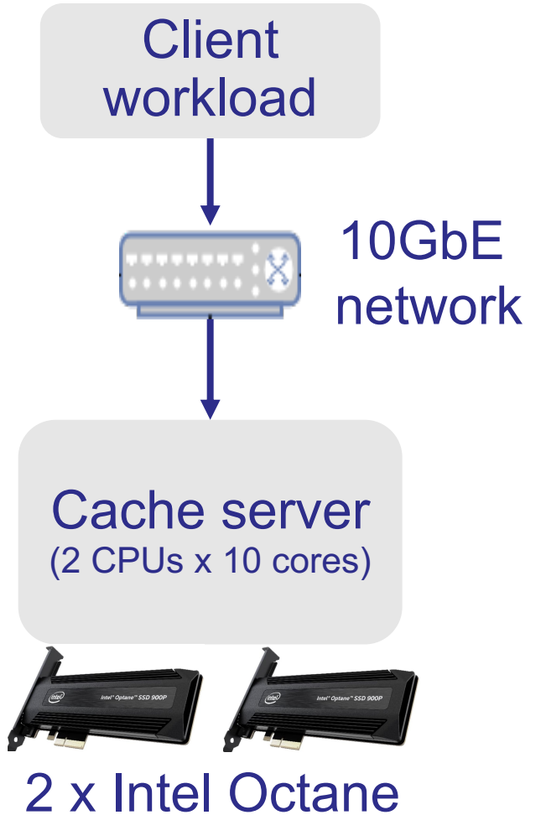
Compare uDepot memcache API vs. DRAM-based memcache systems

Memaslap workload:

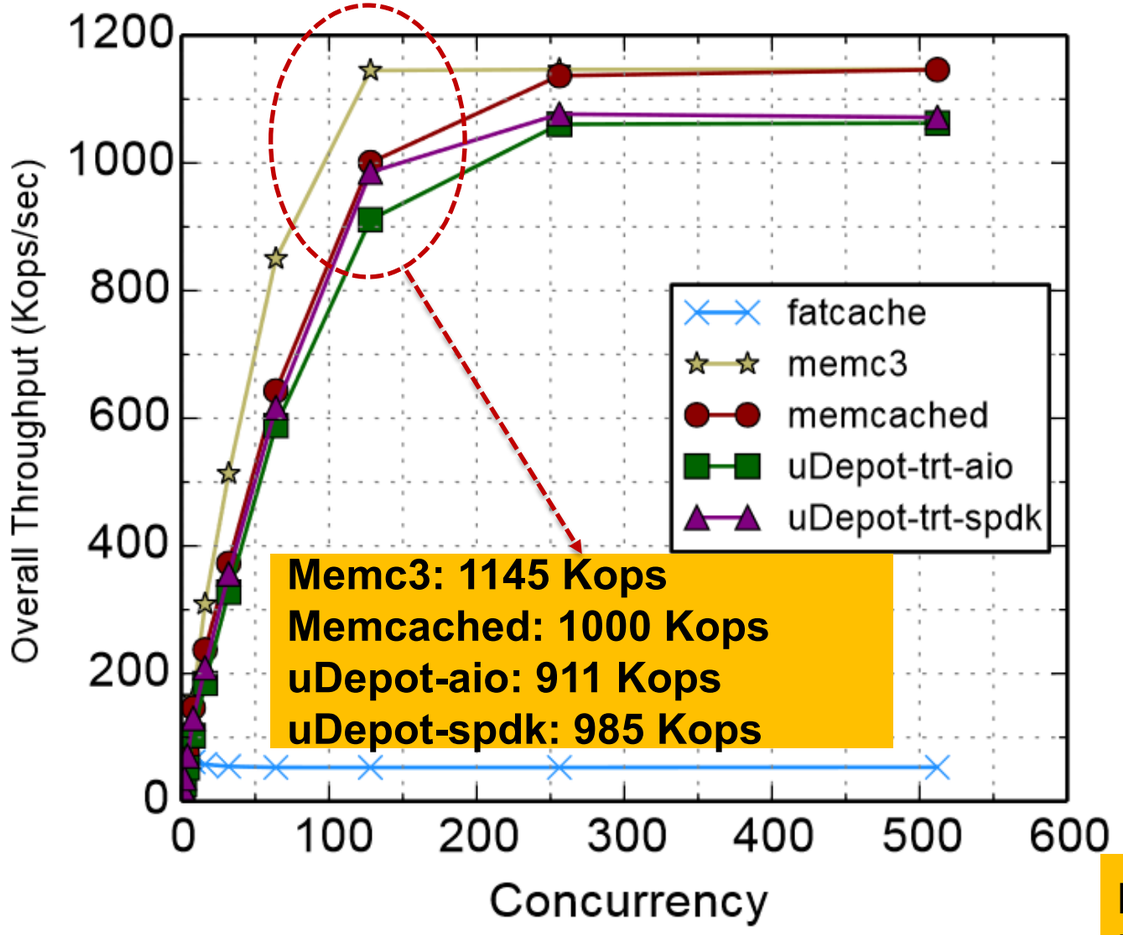
- Standard Memcached benchmark
- 90/10% GET/PUT, 1kB objects

Cache systems:

- Memcached (default system)
- Memc3 (optimized Memcached replacement)
- Fatcache (SSD implementation)



uDepot Memcached API performance



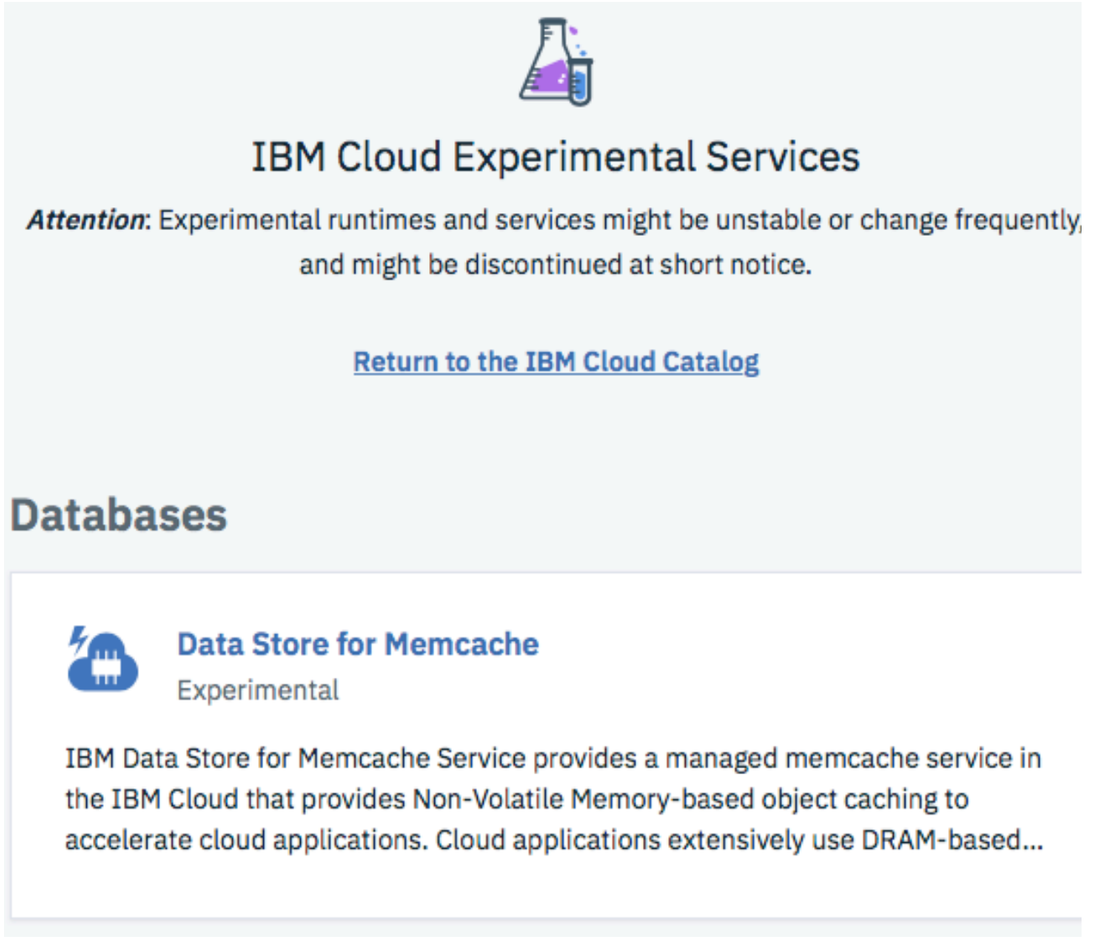
M
M
uDepot-aio: 67us
uDepot-spdk: 51us

memcache-compatible cloud data store

uDepot-based cloud caching service:

- Experimental service on the IBM Cloud
- Memcached drop-in replacement (API compatible)

https://cloud.ibm.com/catalog/services/data-store-for-memcache



Data Store for Memcache

- Cloud and mobile applications use NoSQL caching extensively for performance
  - Response time reduction, throughput elasticity, hotspot elimination, data sharing across apps
- Memcached is one of the most pervasive NoSQL caching systems
  - Client accesses cache capacity on the server over the network
  - Memcached uses **DRAM** as the storage medium
- AWS, MS Azure & Google Cloud all offer managed Memcached services

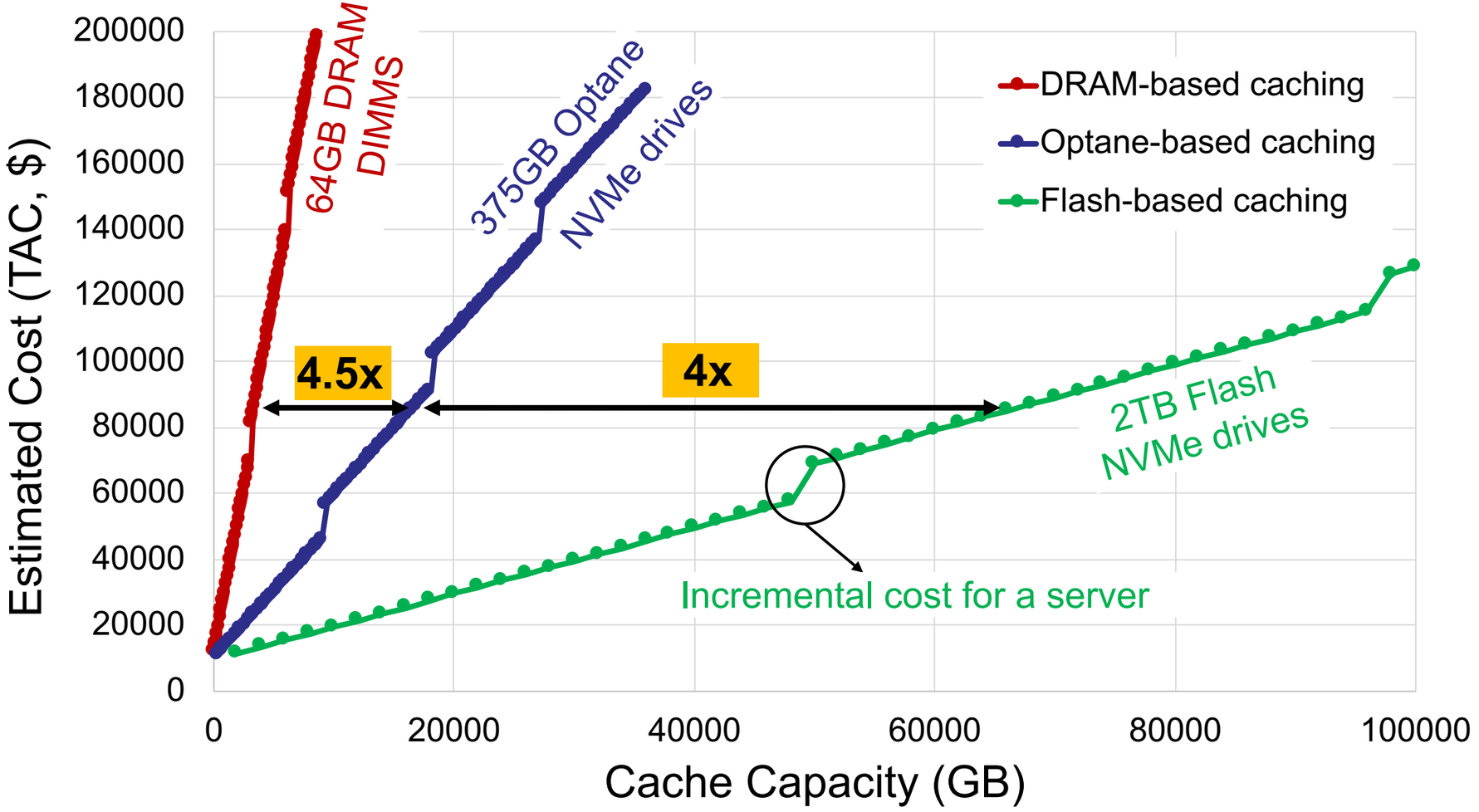
Applications using Memcached

source: stackshare.io

+ 648 more

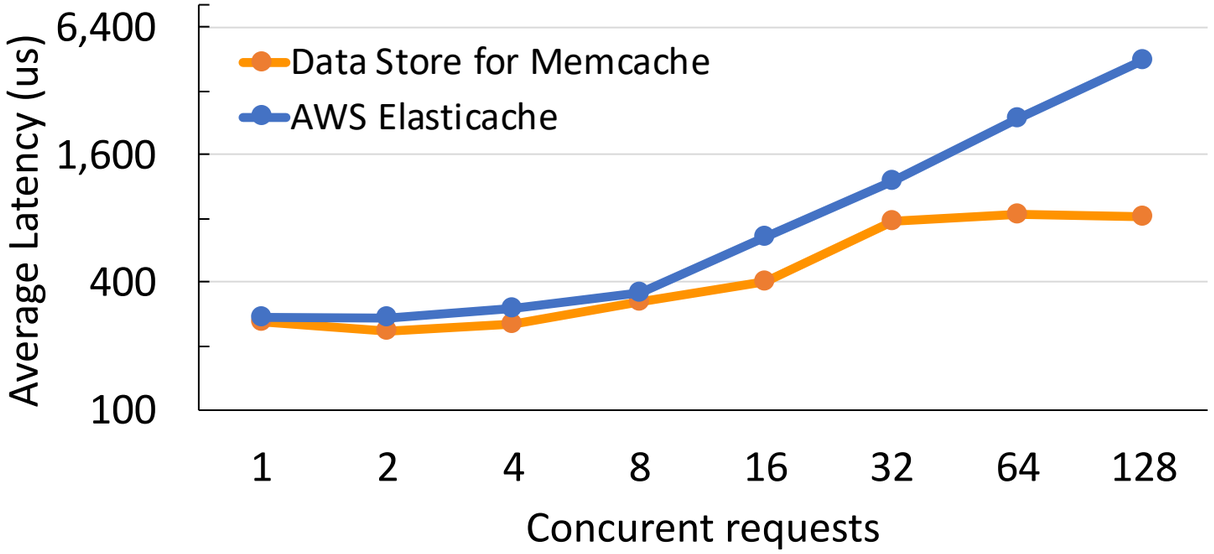
Enterprises using cloud-managed Memcache services

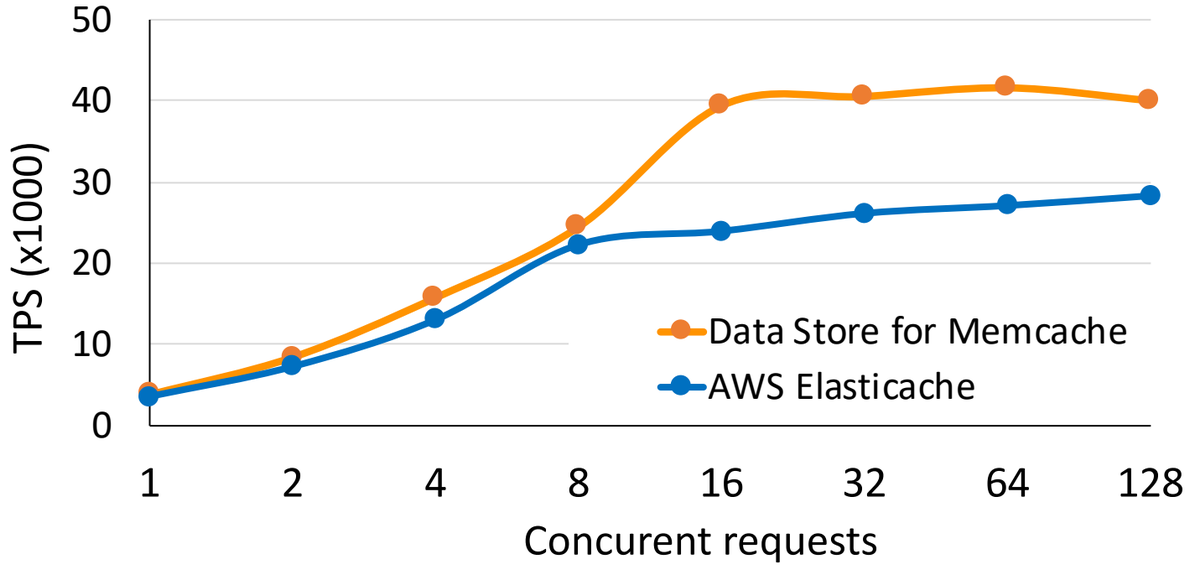
Data Store for Memcached cost



Data Store for Memcache vs. AWS ElastiCache

- Managed memcache services:
  - a) Data Store for memcache (lite plan, free)
  - b) AWS Elasticache (cache.t2.micro, free)
- Experiments within the same availability region (IBM Cloud dal10, AWS us-east2)
- Memaslap 90/10% GET/PUT workload





Competitive performance with DRAM-backed ElastiCache!

Summary

- *Disruptive idea*: Fast NVM devices offer opportunities for **cost reduction** by replacing DRAM, but existing data store technologies **fail** to match their performance.

- *Technology*: **uDepot**, a data-store designed ground-up to deliver the performance of NVM devices to the application.

- *IBM Cloud service*: **Data Store for Memcache** is an experimental IBM cloud service, built using containerized uDepot, that can be used as a drop-in replacement for an ubiquitous DRAM-based cache at a **lower cost** and **similar performance**.

Thank You !

uDepot: "*Reaping the performance of fast NVM storage with uDepot*",
Kourtis et al., FAST'19.

SALSA: "*Elevating commodity storage with the SALSA host translation layer*",
Ioannou et al., MASCOTS'18.

More information at: *www.zurich.ibm.com/cci*