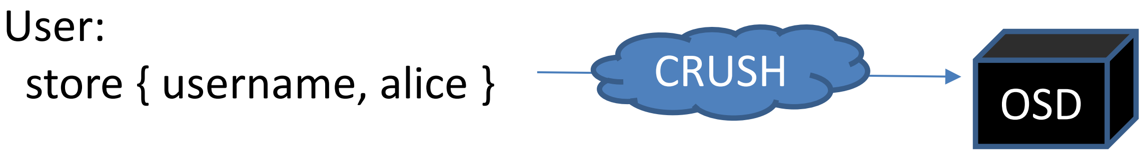
KvsStore: CEPH Object Store for Key-Value SSDs

Memory Solution Lab
Yangwook Kang, Pratik Mishra, James Li, Yangseok Ki
10/24/2018

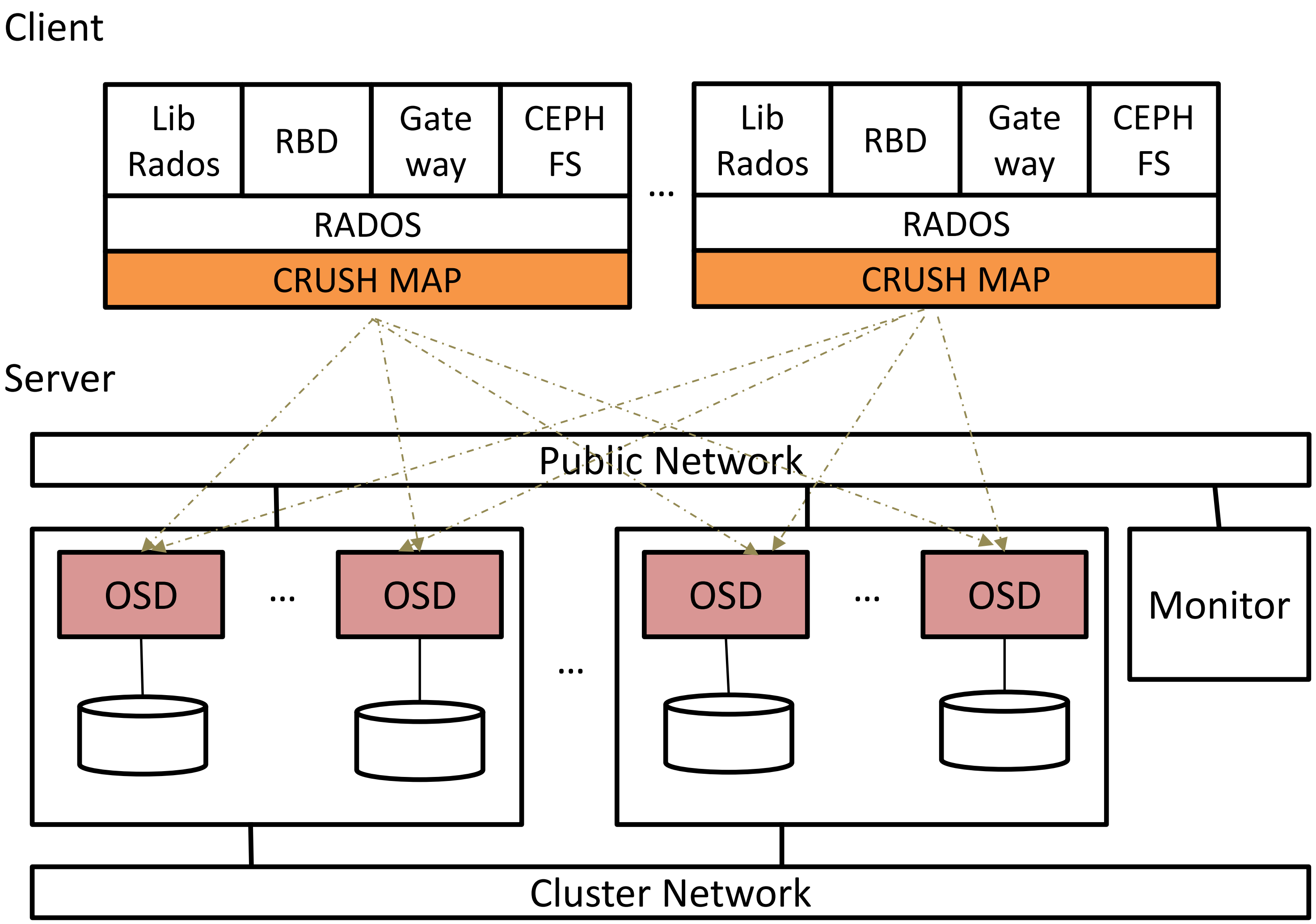
CEPH

- **An object-based distributed storage system designed to provide high scalability and strong consistency**
- **A de-facto standard distributed storage backend for open stack**

- **Main Features**
  - CRUSH: a stateless object distribution algorithm
  - OSD: a self-managed storage node with an object interface



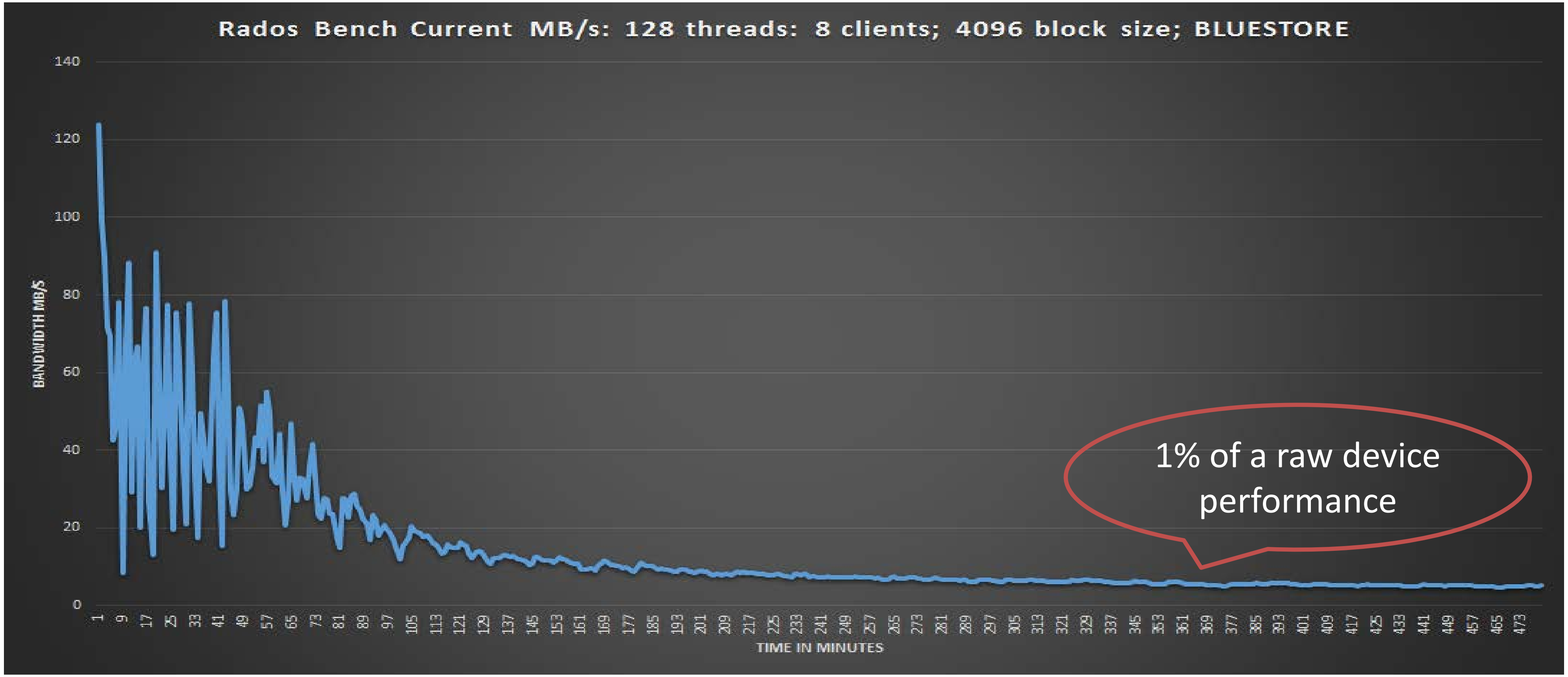
CEPH Overview



- The throughput of each OSD determines the overall performance
- The resource utilization of each OSD influences an in-node scalability
- Inter-node scalability is determined by an efficiency and fairness of CRUSH

Performance of CEPH

- Overall throughput of Ceph is 1-10% of the device performance



- The recent report from Micron shows that the sustained performance can go up to 10% of the device performance with Intel Purley processors (110MB/s, 10 FIO clients) https://www.micron.com/about/blogs/2018/may/ceph-bluestore-vs-filestoreblock-performance-comparison-when-leveraging-micron-nvme-ssds

Problems with Underutilized Disks in Ceph

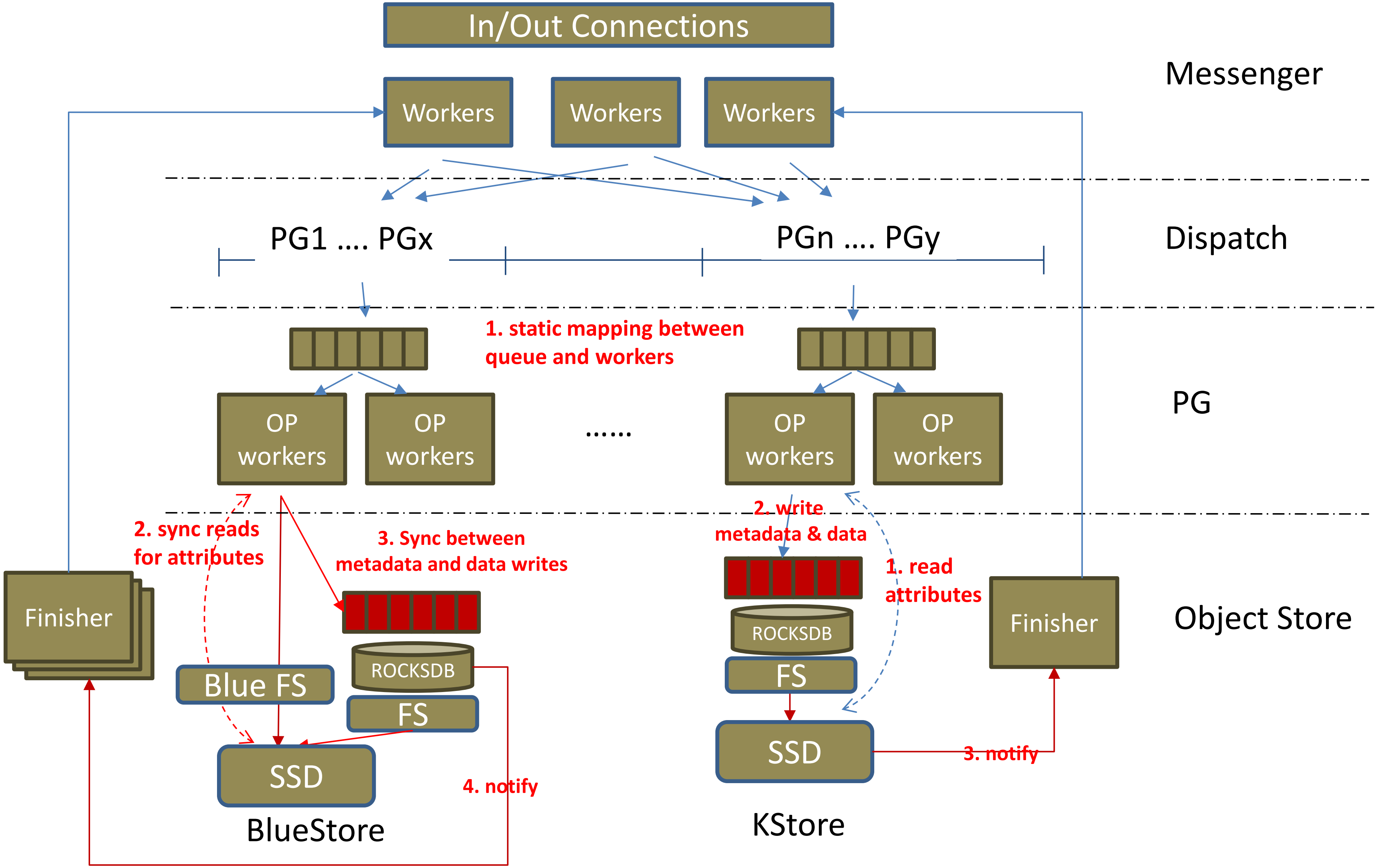
- Ceph OSD cannot get benefit from high performance storage devices such as NVMe SSDs

- Unbalanced system resource usage
  - CPUs are busy while disks are idle
    - More than 20 threads are running concurrently per OSD
  - More storage nodes are needed for a better performance

How can we improve the efficiency of OSD so that the gap between the overall throughput and the device performance can be minimized?

Where do bottlenecks occur? (1)

CEPH OSD Architecture

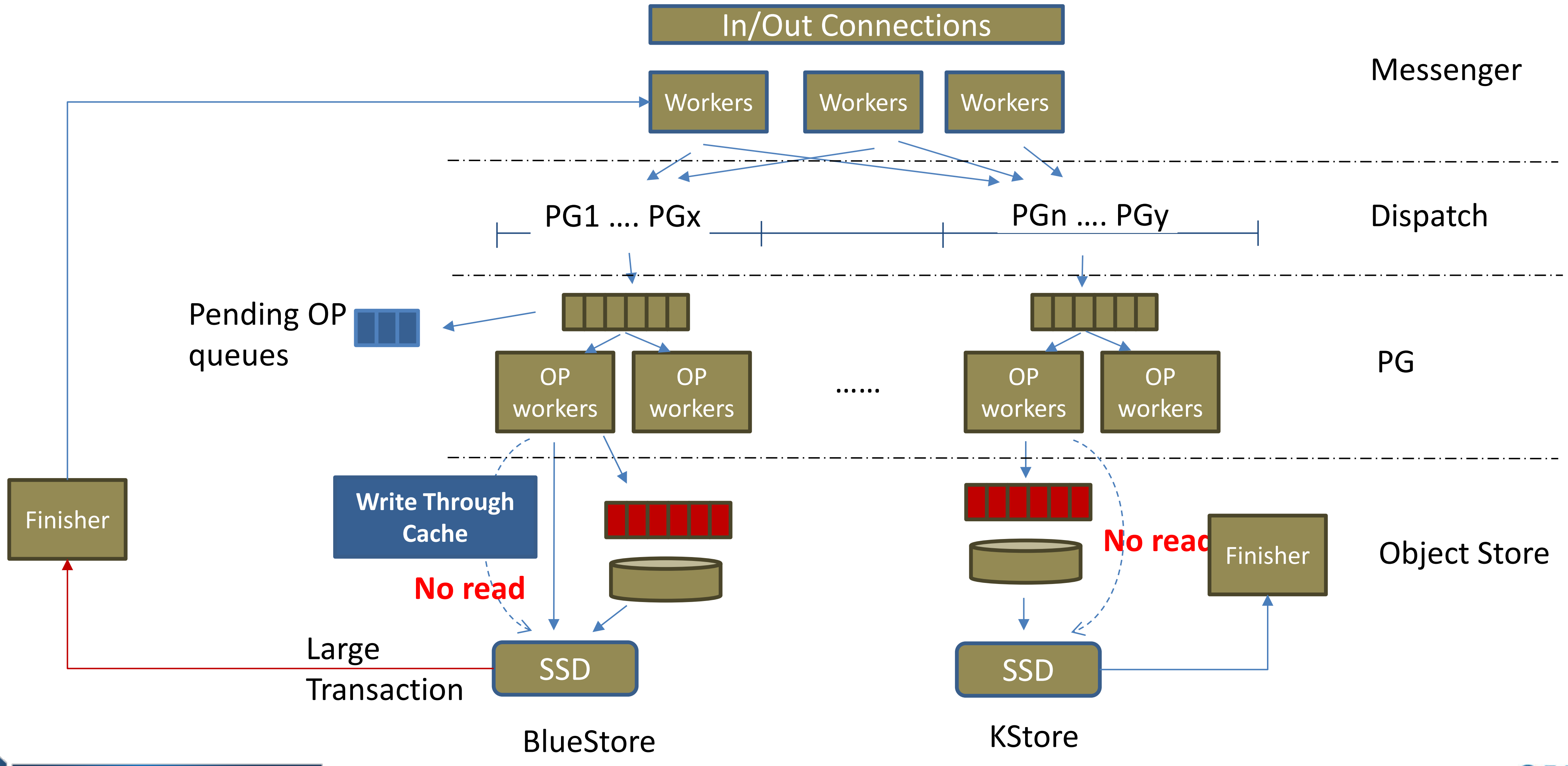


Where do bottlenecks occur? (2)

- **Multiple attribute reads before writes**
  - 2 object attributes are synchronously read before each write
- **Use of large batch operations**
  - Large batch I/Os increase latency and slow down the I/O notification
  - Due to the strong consistency requirement requires, clients need to wait, holding the requests while a large batch is processed and notified
- **Synchronization between data and metadata writes**
- **Use of host-side key-value stores**
  - Host-side key-value stores require lots of CPU and memory resources
  - High compaction overheads -> performance variations
- **Job distribution between request queues and workers**
  - A fixed number of workers are associated with each request queue (Shard)
  - Concurrency can be limited based on the ratio between number of Shards and PG

Existing Approaches (1)

Existing approaches provide a partial solution to this problem



Existing Approaches (2)

- **Pending OP Queues**
  - Solve the issue where one PG worker can block other workers in the same Shard
  - Two lookups requiring an access to an additional queue every time

- **Write-through cache**
  - Maintaining a write-through cache for attributes requires a huge amount of memory
    - A couple of 16B key-256B value pairs per key

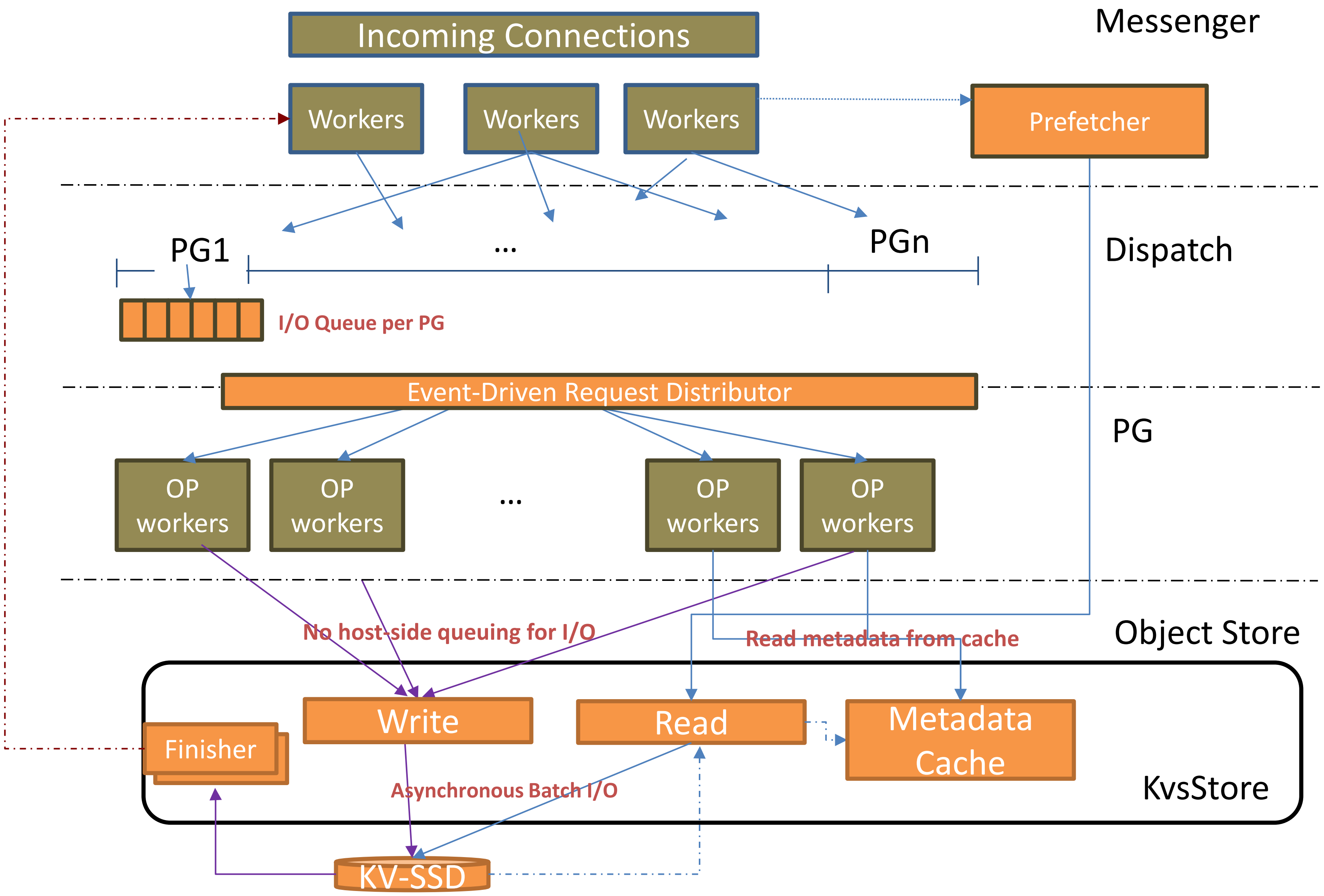
    (272B * 1,000,000,000 keys => 253 GB of memory per device)
    - It can severely hurt the scalability of the system

Our Approach

- Offload host-side key-value management to a underutilized storage devices
  - Eliminate the need for host-side key-value stores
- Use an event-driven scheduler, replacing the need for pending OP queues
- Data path optimization
  - Use a device I/O queue directly
  - Use a read prefetching to avoid issuing synchronous I/Os

Overall Architecture of CEPH OSD + KvsStore

- Global OP worker pool
- KvsStore
  - Async-batched I/Os
- No on-demand reads

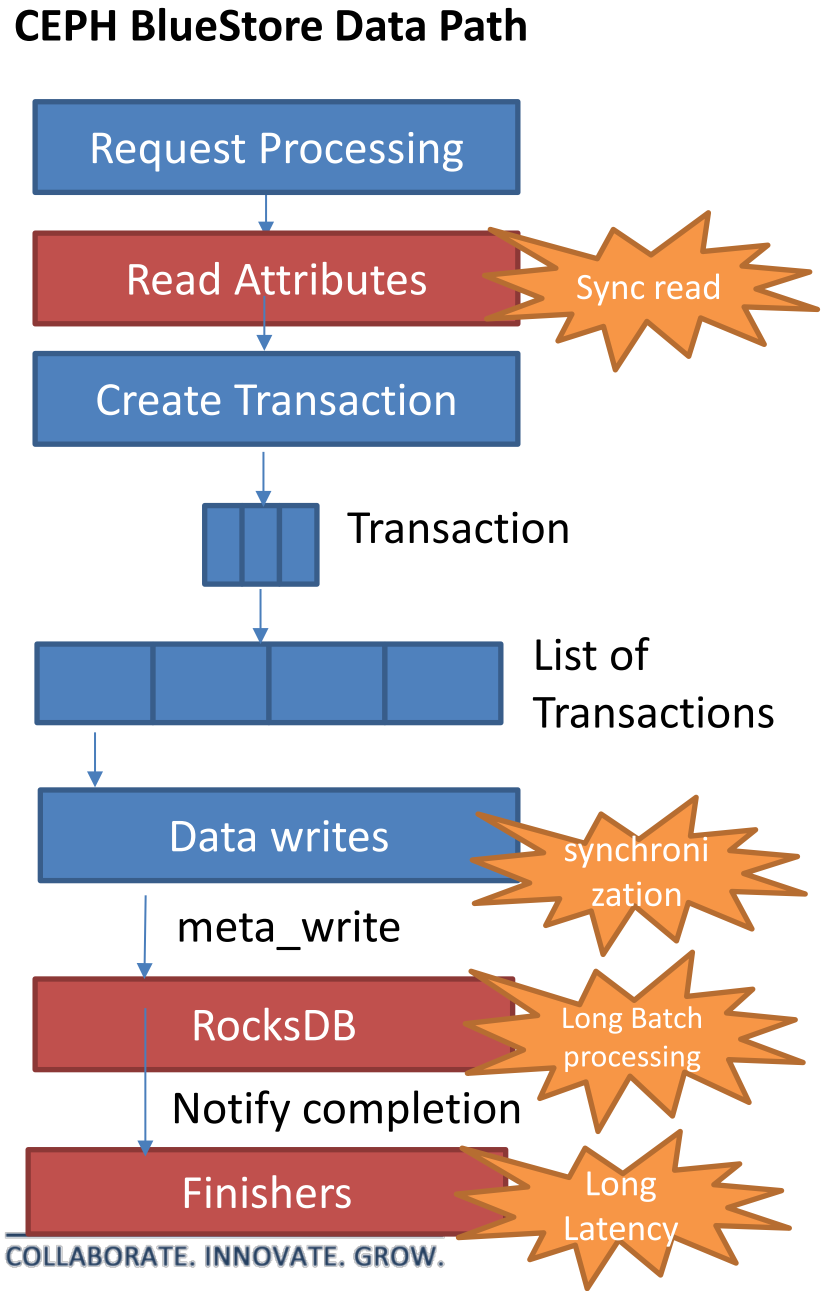


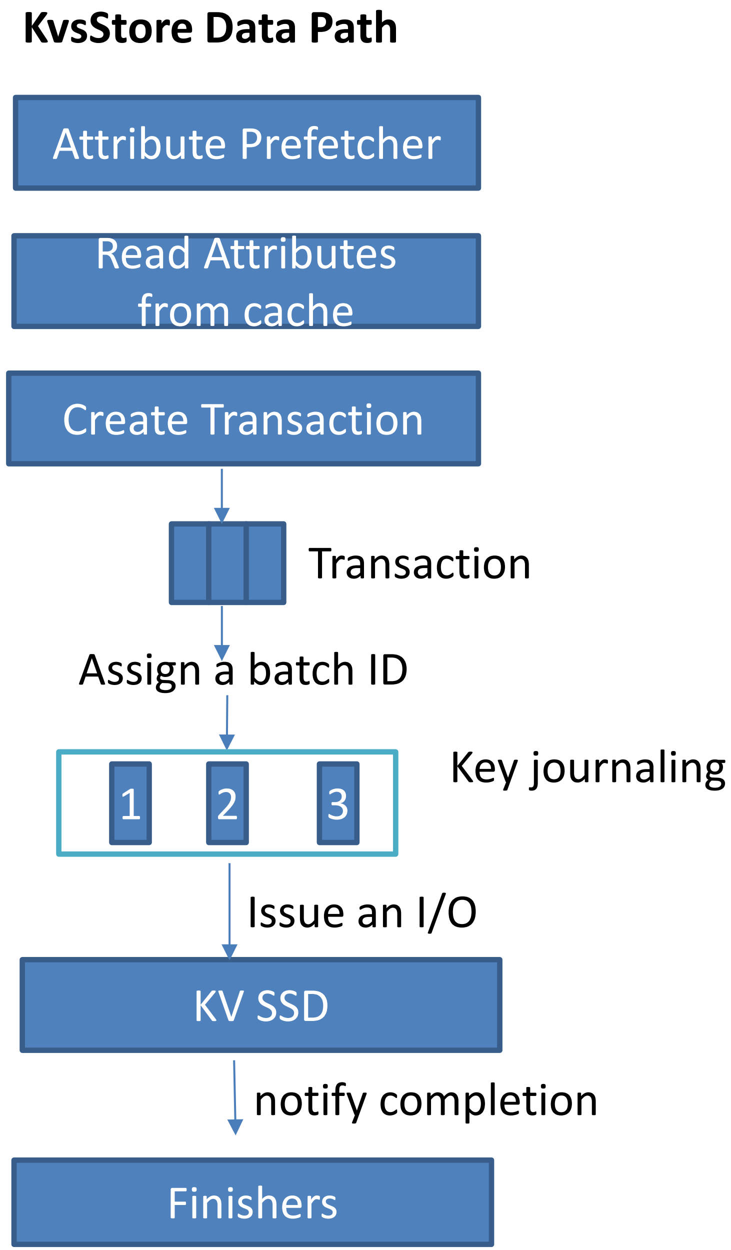
CEPH KvsStore Overview

- **Ceph ObjectStore supports 43 operations on three types of objects:**
  - Objects : user-provided key-value pairs
  - Attributes: small key-value pairs
  - OMAP: large key-value pairs

- **Design Choices**
  - Write operations
    - Each operation is converted to a single KV device I/O operation (exploiting low read/write latency of KV-SSDs)
    - All requests are issued asynchronously
  - Read operations
    - I/O is issued asynchronously, but the caller waits for the completion
  - Management operations
    - List operations, such as list_collection are list_omap_entries, are implemented using iterators
    - OSD metadata is written to a file system
  - Write order
    - Since device operations can be executed out-of-order, we keep track of the write order and force it before sending the response

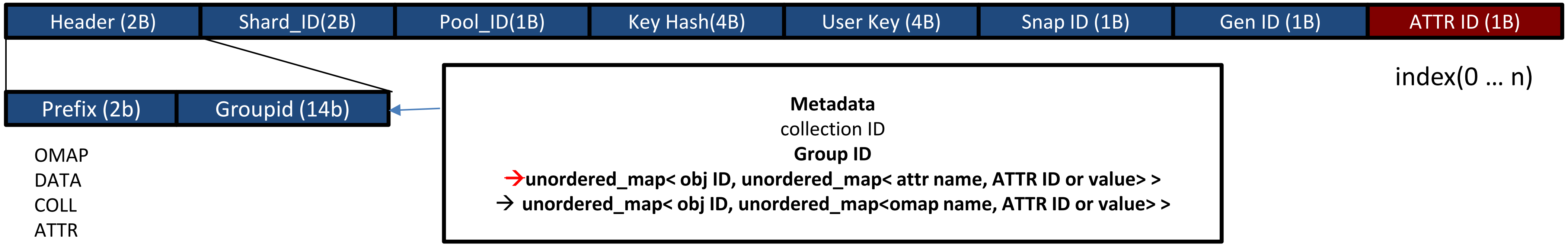
KvsStore Design: Prefetching & Asynchronous Batch Operations





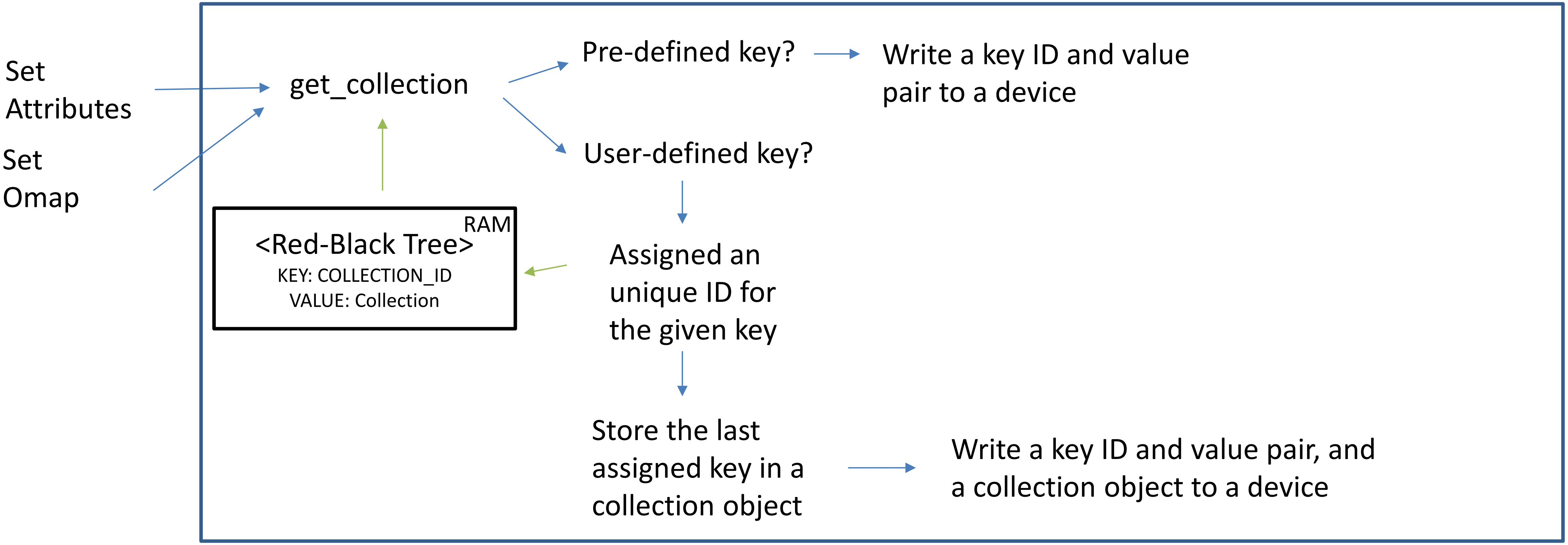
KvsStore Design: Key Structure

- *Since KV-SSD currently only supports 16B key, we reduced the size of each information and encoded it in the key*
- *Attribute names and OMAP entry names are treated specially*
  - *Pre-assigned / Dynamically-assigned*
  - *Unknown names are stored in a metadata object*



KvsStore Design: Metadata Management

- *CEPH metadata is stored as object attributes and OMAP entries*
  - *KvsStore converts them as individual key-value requests to avoid buffering*



I/O Handling in CEPH

Initializing KV SSD

- open KVSSD, open *namespace*, create a *submission* queue, and create a *completion* queue

```
kv_result ADI::open(std::string &devicepath, int queuedepth) {

  kv_result ret = KV_SUCCESS;

  this->devH = NULL;
  this->nsH = NULL;

  kv_device_init_t dev_init = { devicepath.c_str(), "/tmp/kvssd_emulator.conf", FALSE, TRUE };

  ret = kv_initialize_device(&dev_init, &this->devH);
  if (ret != KV_SUCCESS) return ret;

  ret = get_namespace_default(this->devH, &this->nsH);
  if (ret != KV_SUCCESS) { kv_cleanup_device (devH); devH = 0; return ret; }

  // create a submission/completion queue

  const int cqid = _create_queue(queuedepth, COMPLETION_Q_TYPE, &this->cqH, 0);
  _create_queue(queuedepth, SUBMISSION_Q_TYPE, &this->sqH, cqid);

  //derr << "KVSSD " << devicepath << " is opened successfully" << dendl;
  return ret;
}
```

Open a device

Open a namespace

Open queues

Submit KV I/Os

- Write I/O requests to a Transaction is submitted to KV-SSD asynchronously
- When the device queue becomes full, it tries again with an increasing delay

```
int ADI::submit_batch(aio_iter begin, aio_iter end, void *priv)
{
  int attempts = 16;
  int delay = 30;

  aio_iter cur = begin;
  while (cur != end) {

    kv_postprocess_function f = { write_cb, priv };
    kv_result res = kv_store(sqH, nsH, cur->first, cur->second, KV_STORE_OPT_DEFAULT, &f);
    if (res == KV_ERR_QUEUE_IS_FULL && attempts-- > 0) {
      usleep(delay);
      delay *= 2;
      continue;
    }

    if (res != KV_SUCCESS) {
      return res;
    }

    ++cur;
  }

  return KV_SUCCESS;

}
```

callback function

I/O context

Namespace handle

A key-value pair

Submission queue handle

Processing I/O completion

- ***kv_io_context*** **contains the information about the completed I/O including key, value, size, and etc.**
- ***private_data*** **contains a user-provided pointer that can handle the completion**

```
void write_callback(kv_io_context *op) {
  KvsTransContext *txc= (KvsTransContext *)op->private_data;
  if (!txc) { ceph_abort();  };

  txc->aio_finish(op);
}
```

- **In case of synchronous I/Os, submit I/O asynchronously and let the caller wait for a completion**
  - ***private_data*** **contains the struct ioevent that has a mutex and a condition variable**

```
struct ioevent {
    bool finished;
    std::mutex io_lock;
    std::condition_variable io_cond;
    kv_result retcode;
    CephContext *cct;
    ioevent(CephContext *c = 0) : finished(false), retcode(KV_SUCCESS), cct(c) {}

    void set_finished() {
      std::unique_lock<std::mutex> l(io_lock);
      finished = true;
      io_cond.notify_one();
    }

    void block_if_not_finished() {
      std::unique_lock<std::mutex> l(io_lock);
      while (!finished) {
        io_cond.wait(l);
      }
    }
};
```

Called when I/O is finished

Pause the thread until I/O is finished

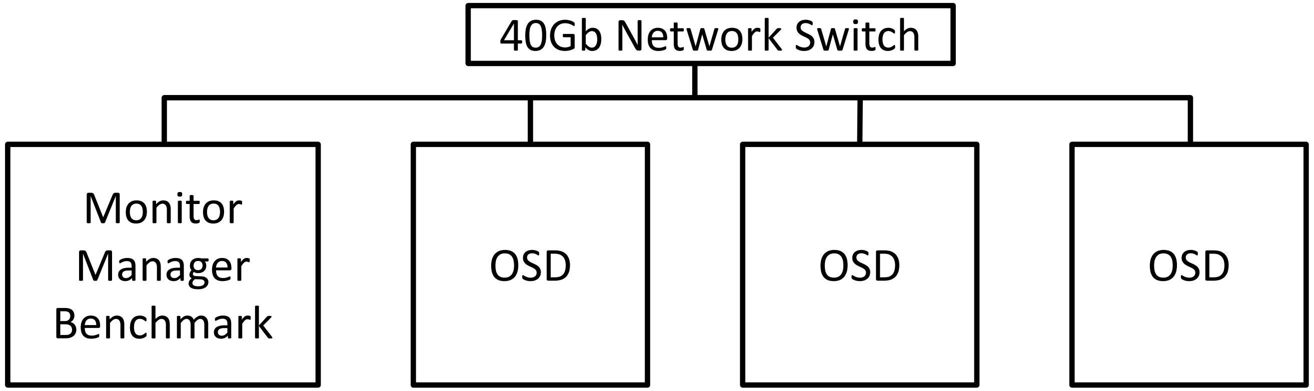
COLLABORA

SAMSUNG

Running CEPH

Experimental Setup

- **CEPH storage server consists of**
  - monitor daemons
  - manager daemons
  - n OSD nodes
- **Benchmark**
  - Rados bench
- **Our Cluster Configuration**
  - CPU: Intel E5-2695 @2.1 Ghz (36 cores with hyper-threading)
  - RAM: 128GB
  - Device: PM983 KVSSD
  - Network: 40Gb Ethernet



Installation

- **Operating system:**
  - Ubuntu 16.04, ext4, kernel version: 4.9.5

- **KVSSD APIs and Drivers:**
  - https://github.com/OpenMPDK/KVSSD

Building and Running CEPH (1)

- **Install dependencies**
  - sudo ./install-deps.sh

- **Cmake**
  - cd ./build
  - rm –rf ceph-runtime && mkdir –p ceph-runtime
  - cmake -DWITH_TESTS=OFF -DWITH_FIO=ON -DFIO_INCLUDE_DIR=../fio -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=./ceph-runtime ..

- **Build**
  - Make
  - Make install

Building and Running CEPH (2)

- **Running**
  - Step 1. Load kernel drivers for KVSSD and format
  - Step 2. Load Monitor/Manager/OSD servers
  - Step 3. Run benchmarks

Step 1. Load kernel drivers for KVSSD and format

- **Download**
  - git clone https://github.com/OpenMPDK/KVSSD
  - cd PDK/driver/PCIe/kernel_driver/kernel_v4.9.5

- **Compile**
  - make

- **Reload the nvme driver**
  - rmmod nvme
  - rmmod nvme_core
  - insmod nvme-core.ko
  - insmod nvme.ko

Step 2. Load and Deploy CEPH Daemons

- **The procedure to run CEPH daemons**
  - http://docs.ceph.com/docs/mimic/start/

- **This procedure includes the following steps**
  - Terminate any remaining OSD processes
  - Setup remote deploy directories
  - Deploy CEPH binary to the remote nodes
  - Format devices
  - Start monitor daemon
  - Start manager daemon
  - Start OSD daemons in the remote servers

Step 2. Setup Remote Directories

```
[10.10.10.12] formatting devices
[10.10.10.13] formatting devices
[10.10.10.14] formatting devices
Success formatting namespace:1
Success formatting namespace:1
Success formatting namespace:1
Step : Devices formatted
[10.10.10.11] sudo rm -rf /mnt/nvmeceph/ceph-runtime
[frombuild] deploying ceph-runtime to 10.10.10.11/mnt/nvmeceph/ceph-runtime
[10.10.10.12] sudo rm -rf /mnt/nvmeceph/ceph-runtime
[frombuild] deploying ceph-runtime to 10.10.10.12/mnt/nvmeceph/ceph-runtime
[10.10.10.13] sudo rm -rf /mnt/nvmeceph/ceph-runtime
[frombuild] deploying ceph-runtime to 10.10.10.13/mnt/nvmeceph/ceph-runtime
[10.10.10.14] sudo rm -rf /mnt/nvmeceph/ceph-runtime
[frombuild] deploying ceph-runtime to 10.10.10.14/mnt/nvmeceph/ceph-runtime
Step : deploy_dir completed
[10.10.10.11] Directory path set to /mnt/nvmeceph/ceph-deploy
[10.10.10.11] Remove directory /mnt/nvmeceph/ceph-deploy
[10.10.10.11] mkdir -p /mnt/nvmeceph/ceph-deploy/mon
[10.10.10.11] mkdir -p /mnt/nvmeceph/ceph-deploy/out
[10.10.10.12] Directory path set to /mnt/nvmeceph/ceph-deploy
[10.10.10.12] unmounting devices
[10.10.10.12] Remove directory /mnt/nvmeceph/ceph-deploy
[10.10.10.12] mkdir -p /mnt/nvmeceph/ceph-deploy/mon
[10.10.10.12] mkdir -p /mnt/nvmeceph/ceph-deploy/out
directory structure created
osd directory created
[10.10.10.13] Directory path set to /mnt/nvmeceph/ceph-deploy
[10.10.10.13] unmounting devices
[10.10.10.13] Remove directory /mnt/nvmeceph/ceph-deploy
[10.10.10.13] mkdir -p /mnt/nvmeceph/ceph-deploy/mon
```

- **ceph-runtime**
  - CEPH binary
- **ceph-deploy**
  - CEPH configuration
  - Log files

Step 2. Starting Monitor, Manager, and OSD

- **Monitor**
  - register new daemon to a keyring
  - ceph-mon -mkfs
  - ceph-mon -i hostname

- **Mgr**
  - create a manager with a name, e.g. ‘sam’
  - ceph-mgr -I sam

- **OSD**
  - Register new OSD to a keyring
  - ceph-osd –mkfs
  - ceph-osd –I OSDID

Step 2. Checking the status of CEPH

ceph –s

```
osd running
  cluster:
    id:     5ee5808d-eb80-4964-90ed-58cd96d2e8cd
    health: HEALTH_OK

  services:
    mon: 1 daemons, quorum CephDev11
    mgr: sam(active)
    osd: 3 osds: 3 up, 3 in

  data:
    pools:   1 pools, 200 pgs
    objects: 0 objects, 0 bytes
    usage:   21121 MB used, 3519 GB / 3539 GB avail
    pgs:     200 active+clean

ceph is currently running
sdjump@CephDev11:/mnt/nvmeceph/ceph-admin$
```

Step 2. Development Environment

- **CEPH provides a vstart.sh**
  - Runs all daemons in a local system

- **We provide scripts to automate the deploy process**
  - setup_kvsstore_clusters.sh [num_of_osd_nodes]
  - setup_bluestore_clusters.sh [num_of_osd_nodes]
  - run_rados_write.sh
  - kill_ceph_pids.sh

Step 3. Benchmark

- **Prepare a pool & setup replication**
  - ceph osd pool create rbd 100
  - ceph osd pool application enable
  - ceph osd pool set rbd size 1
  - ceph osd pool set rbd min_size 1

- **Run Rados bench**
  - sudo ./bin/rados bench -p rbd -b 4096 --max-objects 100000 --run-name m -t 64 30 write --no-cleanup

CEPH Configuration

- **Location of A Configuration File**
  - vstart creates one in the current directory
  - Our scripts creates on in the ceph-deploy directoy

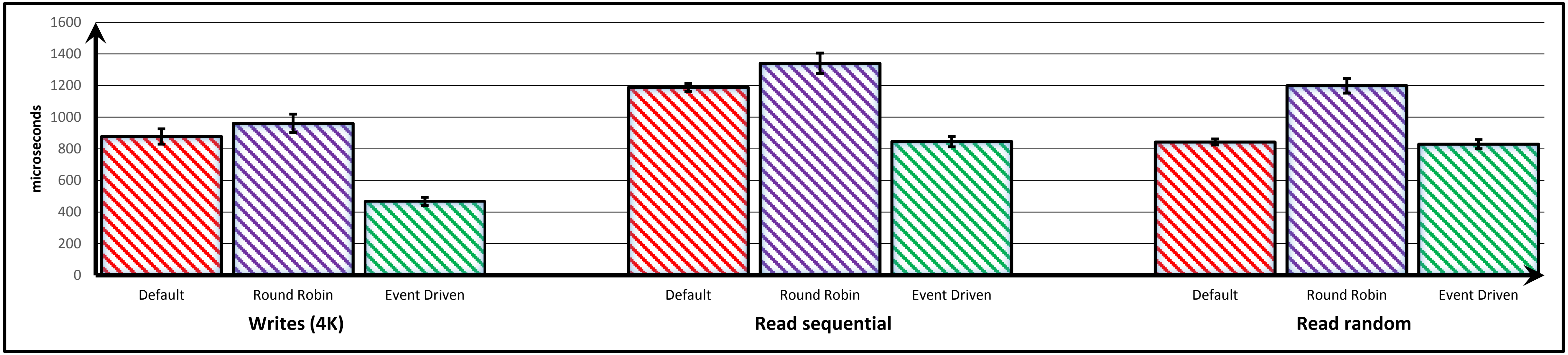
- **Change a type of Object Store**
  - KvsStore is implemented as a type of an object store
  - osd objectstore = bluestore / kvsstore

- **Object Store-specific options**
  - search for bluestore_xxx or kvsstore_xxxx

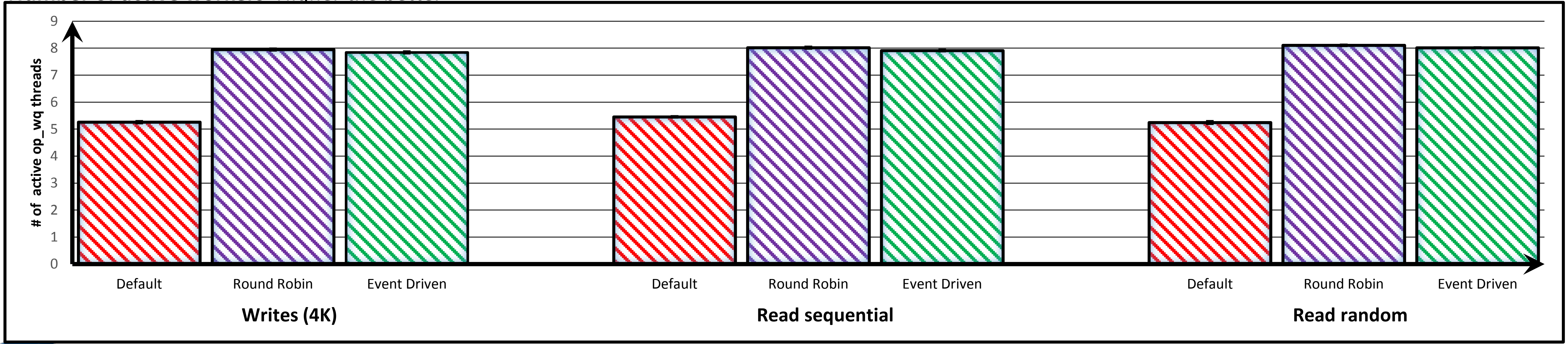
Evaluation

Benefits of Event-Driven Scheduler

Avg. PG queue processing time in cluster - *Lower the better*

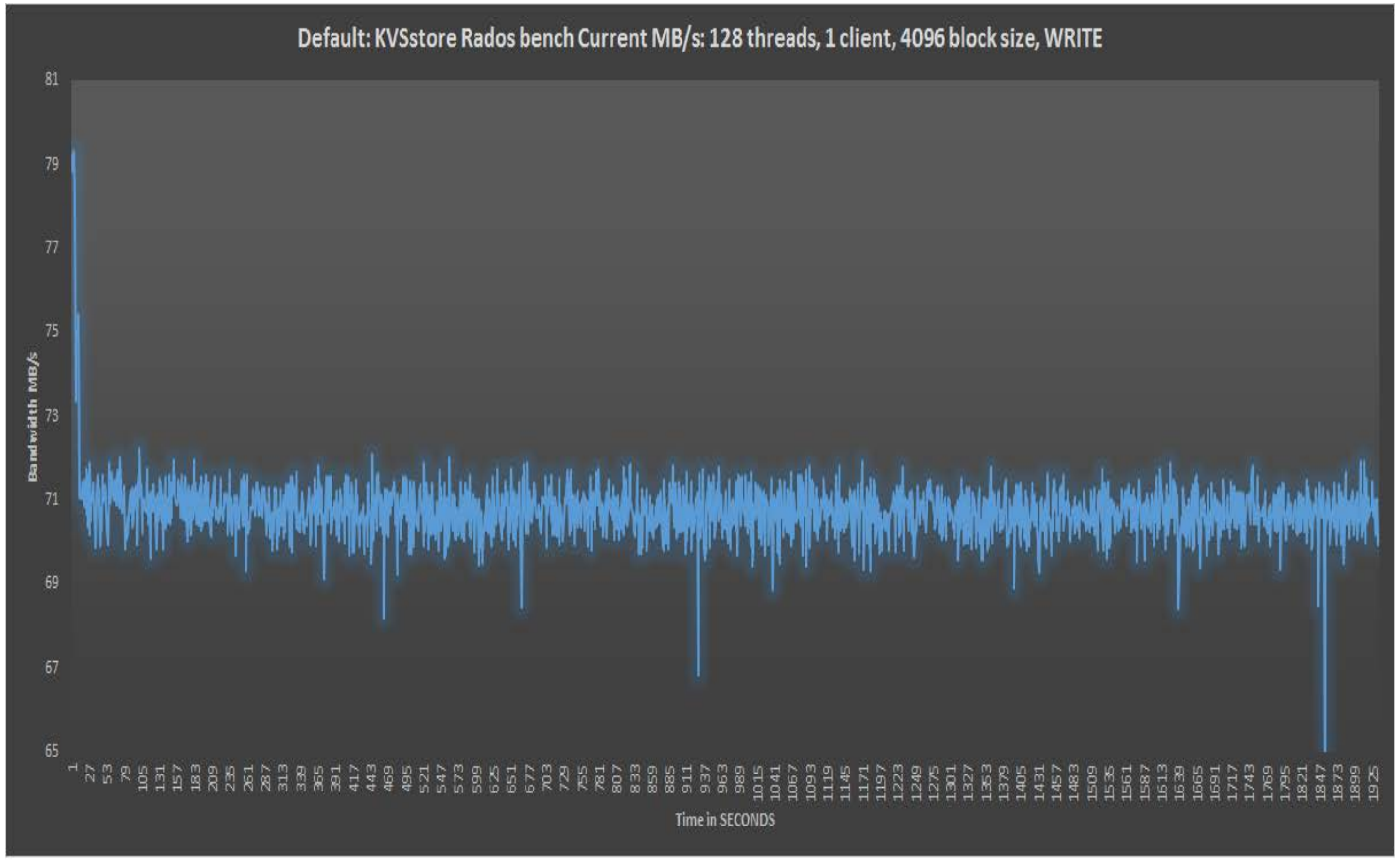


Number of active workers- *Higher the better*

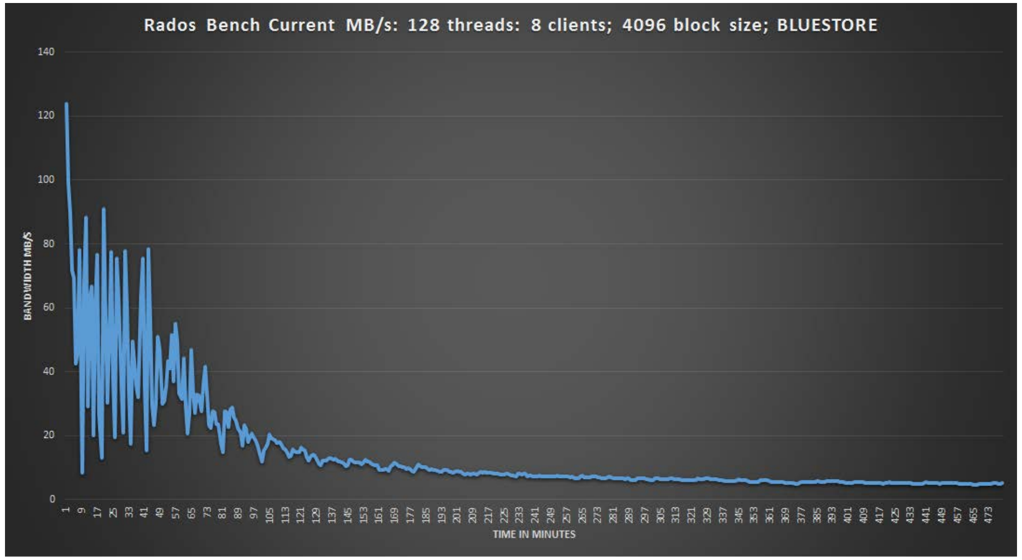


Performance of KvsStore

KvsStore



BlueStore

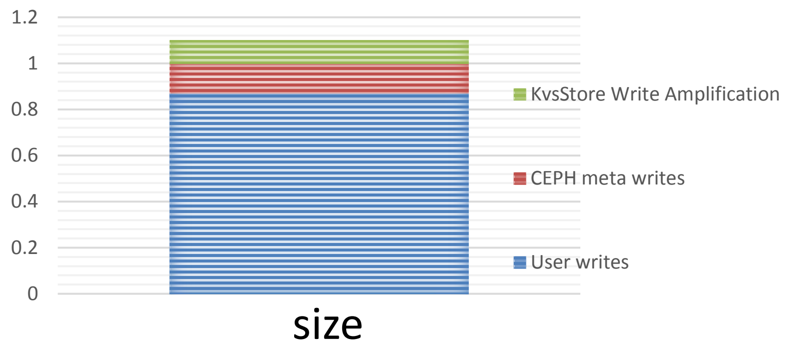


I/O Characteristics of KvsStore

- **Internal I/O Efficiency**

- KvsStore draws the **75% of the device performance**
- Current performance is bounded by the device performance, not CPU anymore

- **Write Amplification**



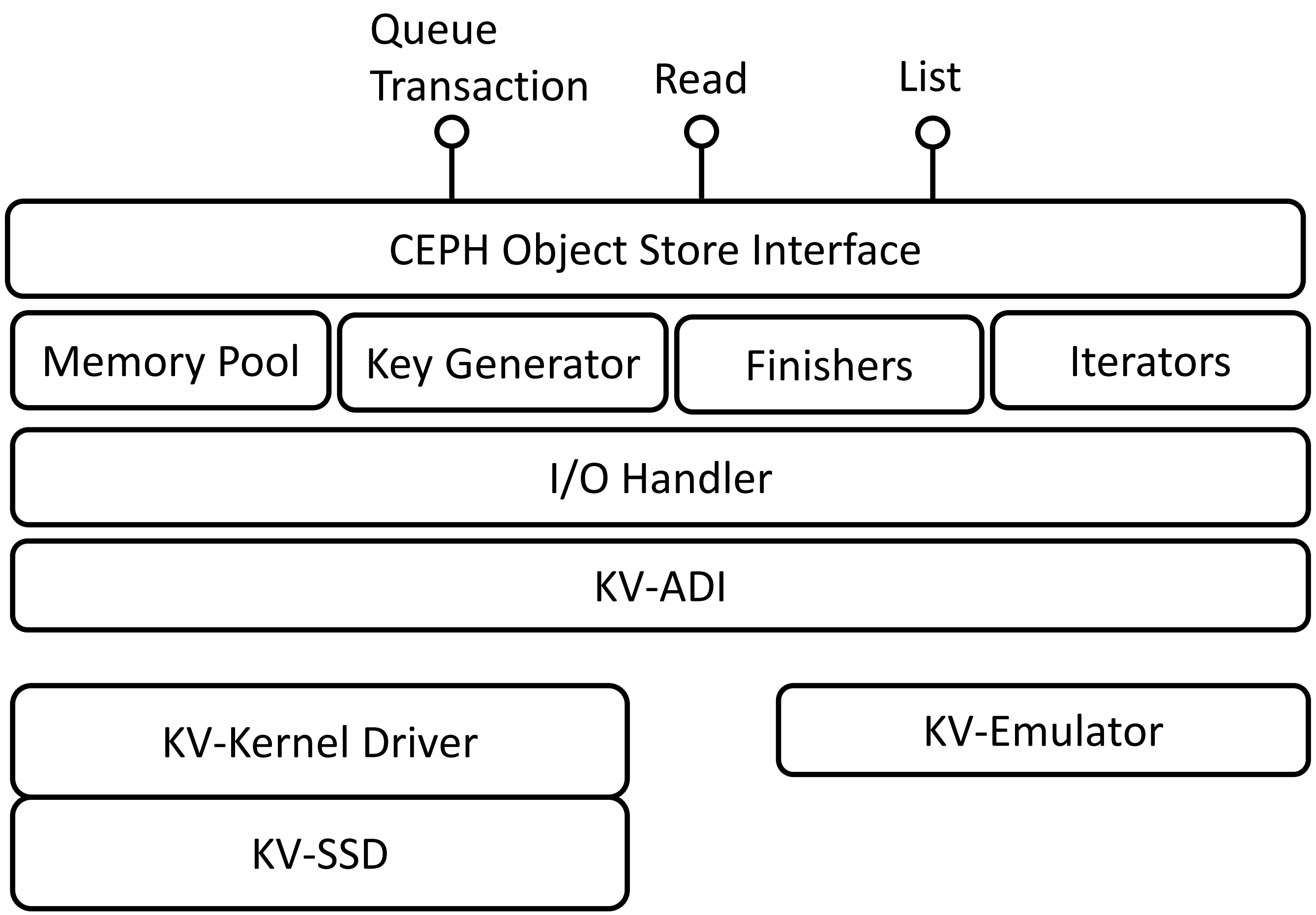
Conclusion

- Event-driven request scheduler
  - Low-overhead request scheduler that improves the processing latency by 20-40%

- KvsStore
  - Replaces a resource hungry host-side key-value stores with Samsung KV-SSDs
  - Provides a 4x better sustained performance than BlueStore
  - Improves the underlying device utilization of CEPH up to 75%

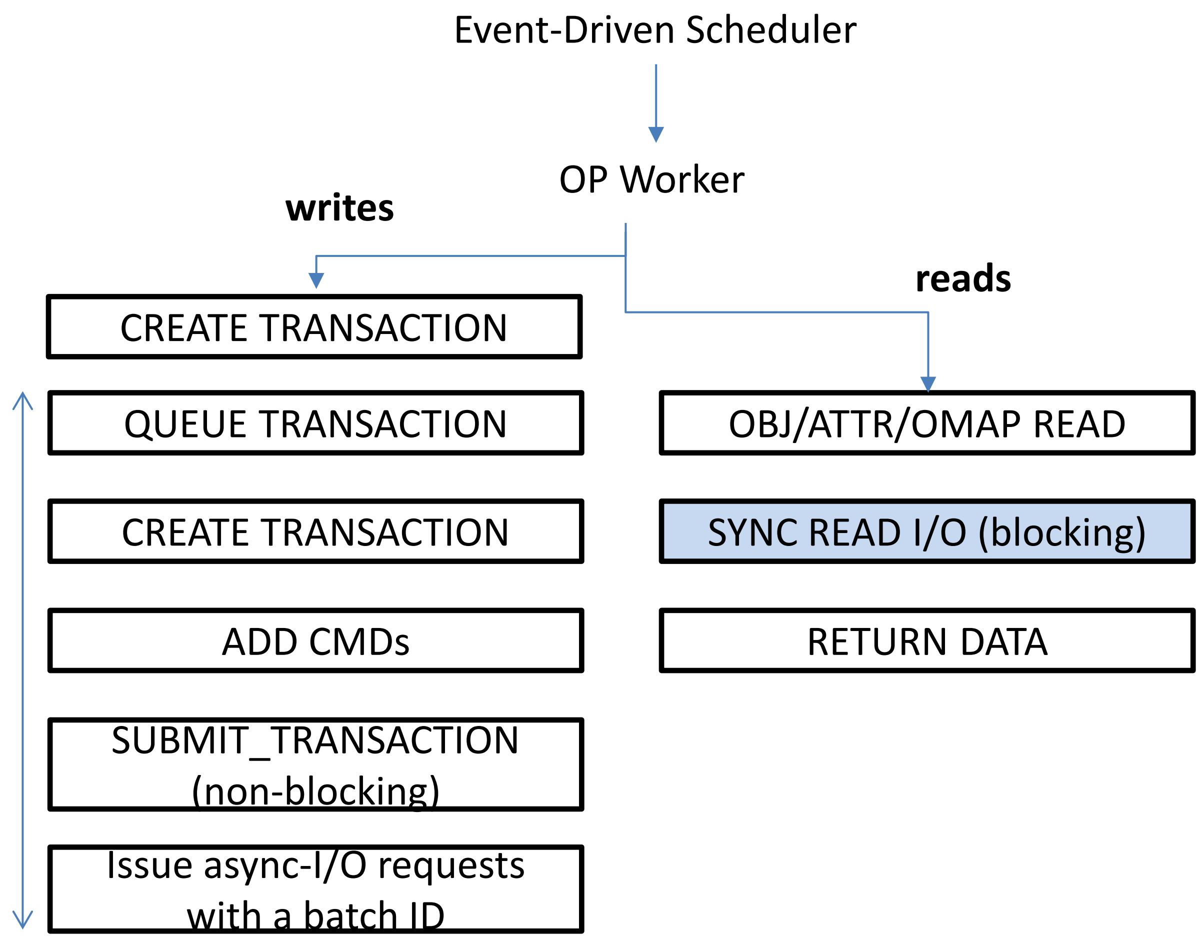
Thank you

Structure of CEPH KvsStore



Life-cycle of I/O Requests in KvsStore

Current Version



With Prefetcher support (In Development)