ACCELERATING REDIS WITH INTEL® OPTANE™ DC PERSISTENT MEMORY
Agenda

- Redis Overview
- Intel® Optane™ DC Persistent Memory (DCPMM) Introduction
  - DCPMM Operational Model
  - SNIA program model and PMDK library.
- Redis Volatile- Migrate Data to DCPMM
  - Redis Data Migration Strategy
  - Redis Data Structure Update
- Redis Persistence on DCPMM
  - Redis RDB Snapshot
  - Redis Pointer Based AOF
  - Redis Persistence With libpmemobj (POC)
- Summary
REDIS AND DCPMM INTRODUCTION
Redis Overview

Redis is an in-memory remote database that offers high performance, replication, and a unique data model to produce a platform for solving problems.

High level illustration on how content is stored in Redis
Intel® Optane™ DC Persistent Memory (DCPMM) - Operational Modes

**APP DIRECT MODE**

- Persistent
- High availability / less downtime
- Significantly faster storage

**MEMORY MODE**

- High capacity
- Affordable
- Ease of adoption†

Memory mode is volatile and App Direct mode support persistent capability
DCPMM Program on APP Direct

Legacy Storage API
- No code changes required
- Operates in blocks like SSD/HDD
- Block atomicity

Persistent Memory
- Code changes is required, byte-addressable
- Bypasses FS page cache
- Requires DAX enabled file system “XFS, EXT4, NTFS”
- Fastest I/O path possible

App Direct “Persistent Memory” is the fastest I/O path
Persistent Library PMDK Introduction

PMDK is a series of persistent libraries for applications.
# PMDK Usage Scenarios

<table>
<thead>
<tr>
<th>PMDK Libraries</th>
<th>Description</th>
<th>Use When</th>
<th>Persistence Need Attention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libpmem/librpmem</td>
<td>Low-level library with primitives persistent APIs</td>
<td>Only need low-level optimized primitives API (memcpy, etc.)</td>
<td>Application need manage data consistence (power fail atomic) and recovery For example, power fail during function pmem_memcpy_persist (pmem, “Hello, World!”); pmem may not “Hello, World!”</td>
</tr>
</tbody>
</table>
| Libmemkind              | Explicitly manage allocations from PMEM                   | 1. Different tiers of objects (hot, cold) can be identified  
2. Persistence is not required                                    | PMEM allocation, not persistent                                                                 |
| Libpmemobj/libpmemblk/libpmelog | Transactional object store, memory allocation, transactions | 1. Direct byte-level access to objects is needed  
2. Persistence is required                                            | Maintain persistent meta data, redo log and undo log, performance may be lower. But it can make sure data power fail atomic and can recovery data after power fail. |

1. Application can use libmemkind+libpmem and consider data consistence and recovery mechanism from application.
2. Application can use libpmemobj and focus on optimize persistent performance.
REDIS VOLATILE WITH DCPMM
Redis Data Migration to DCPMM

• Design option #1 - Store all heap data in PMEM, less code change:
  • Big and sequential data access pattern has better performance than small and random data access pattern in PMEM
  • Management data structures are small and usually be accessed randomly
• Design option #2 - Store large heap data in PMEM, close performance with DRAM
  • Only store large value in PMEM (eg. >=64 byte by default)
  • Keep Redis management data structures in DRAM, need customized and optimized data placement strategy for each data type
Redis Data migration strategy

*Note*: value $\geq$ threshold will move to DCPMM, Keys keep in DRAM (metadata: data structures to manage key-value keep in DRAM)
## Redis typical data structures Update

<table>
<thead>
<tr>
<th>Type</th>
<th>Encoding</th>
<th>Value *ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJ_</td>
<td>OBJ_ENCODING_</td>
<td>***********</td>
</tr>
<tr>
<td>STRING</td>
<td>RAW</td>
<td>SDS, length &gt;= threshold move to DCPMM.</td>
</tr>
<tr>
<td>INT</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>EMBSTR</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td>HASH</td>
<td>HT</td>
<td>DICT, field/value &gt;= threshold, move to DCPMM</td>
</tr>
<tr>
<td>ZIPLIST</td>
<td>Data structure changed</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>INTSET</td>
<td>No change</td>
</tr>
<tr>
<td>HT</td>
<td>DICT, field/value &gt;= threshold, move to DCPMM</td>
<td></td>
</tr>
<tr>
<td>ZSET</td>
<td>SKIPLIST</td>
<td>field &gt;= threshold move to DCPMM.</td>
</tr>
<tr>
<td>ZIPLIST</td>
<td>Data structure changed</td>
<td></td>
</tr>
<tr>
<td>LIST</td>
<td>QUICKLIST</td>
<td>No Change. Quicklist node contain ziplist data structure.</td>
</tr>
</tbody>
</table>

Note: With 2-2-1, 40*2 Gbps network, 1k data and SLA<99% latency 1ms; Redis volatile performance with DCPMM is quite close to the DRAM with redis-benchmark. 100% write > 85% performance and 100% read ~100% performance of opensource with DRAM.
Leverage clflush for Cached Sequential Writes*

- LLC WB evictions lead to near random behavior (lower BW). SW recommendation: Do CLFLUSH often enough to avoid LLC evictions.
- DCPMM Redis depend on pmem_memcpy_persistent to call CLFLUSH to reduce the LLC WB evictions.

Note*: Internal Tuning guide with BKMs in bi-weekly PnP report. (link: CLX_AEP PnP Tuning Guide)
Redis Volatile Performance with DCPMM

- Redis with DCPMM write TPS > 85% DRAM TPS @1024 data size within SLA (128 instances)
- Redis with DCPMM read QPS ~100% DRAM QPS @1024 data size within SLA

Note * SLA is 99% latency <1ms
REDIS PERSISTENCY ON DCPMM
Redis Persistence Introduction

Redis Persistence: RDB used for the disaster recovery

Redis Persistence: AOF is more durable than RDB, recover slower
Use DCPMM for Redis persistency

• Design option #1 – Persist everything in PMEM
  • Use libpmemobj to store data and its mgmt structures in PMEM
  • No need AOF and RDB and keep data persistent, consistent and atomic.
  • provide a fast data restore after server reboots.
  • URL: https://github.com/pmem/redis/tree/reserve_publish_poc + (some fix patches)

• Design option #2 – Pointer based AOF
  • Consider Redis has its own persistency mechanism, leverage RBD/AOF to guarantee data integrity
  • Store key in DDR and AOF (same to Open Source Redis), store value in PMEM (for persistency) and only store its pointer in AOF(for recover) Performance:
  • Much better than Open Source Redis AOF (sync=always)
  • URL: https://github.com/pmem/pmem-redis
Redis persistent with libpmemobj w/o AOF (POC)

- Redis string persistent with libpmemobj, see Fig 1
  - No need redis persistent mechanism AOF
  - Persistent key/value in DCPMM
  - keyoid, valoid maintained in a linked list.
- Batch transaction to reduce the undo/redo overhead.
- Restore the database by scanning keyoid, valoid linked list and quickly load the data to redis.

Note: use libpmemobj w/o AOF is only a POC that can keep the data persistent always with the ACID features. Batch update to improve the persistent performance. In multi-instances cases (1 instance/core), value size is 1K with redis-benchmark, persistent with libpmemobj is about 3.x of the opensource AOF always and 5x restore time.
RDB Issue on DCPMM

- RDB forked a child process to save the memory snapshot. The main process can still service the client.
- Data in DRAM can take advantage of COW that mean data update in DRAM will not be visible by forked child process.
- DCPMM is DAX-enabled, the write to DCPMM is shared in different processes that mean data update in DCPMM will be visible for both parent and child process and snapshot will fail.

App Direct Persistent Memory are shared in processes, fork child process cannot save the memory snapshot of that time pointer.
Redis RDB

- A User Mode COW is designed for the RDB snapshot. See Figure 1.
  - D1, D2, D3, D4 will be saved to RDB in disk by the forked process.
  - Not real free the buffer when delete a data in main process (see Step 2: delete D3).
  - Copy real buffer instead pages during update a data in main process (see Step 4: update D4).
  - After fork done and D1, D2, D3, D3 saved, free the data in COW address list (see Step 7).
- User Mode COW show good performance than kernel COW [2LM] . See Figure 2.
1. Redis get a write request from client
2. Redis store the key in the DRAM
3. Redis use memkind to allocate space and store the big value data into PMEM and use libpmem persistent the big value.
4. Redis save the command as AOF in SSD. AOF only store the key and location of big value (DCPMM)

Note: AOF only store the location of value, reduce the throughput of the disk which is always the bottleneck of the AOF and improve the persistent performance. In multi-instances cases (1 instance/core), value size is 1K with redis-benchmark, Pointer based AOF performance is about 3.x of the opensource AOF always.
1. Data generated and request the application to persistent these data.

2. Application will write the data to the SSD (traditional approach) or write the data to the persistent buffer, once write successfully, the atomic producer address will be updated.

3. Once the data in the persistent buffer is big enough, the data will synced to the SSD and the atomic consumer address will be updated at the same time.

Note: If power loss during sync to SSD, the atomic consumer address will not update, the data still exist on the persistent buffer. Since part of data has been written to the SSD, to make the SSD data be atomic, the SSD log file length will combine with the consumer address together to be a 64 bit atomic data.
## Redis Performance (AOF) pba vs Open source

### Test case
- **Data size:** 1024,
- **Workload:** Set
- **AOF dir:** NVMe SSD

<table>
<thead>
<tr>
<th></th>
<th>pointer-based-aof</th>
<th>persistent ring buffer</th>
<th>appendfsync</th>
<th>Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pba</strong></td>
<td>yes</td>
<td>yes</td>
<td>Everysec</td>
<td>Per record</td>
</tr>
<tr>
<td><strong>Open source-volatile</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>source-always</strong></td>
<td>yes</td>
<td>yes</td>
<td>always</td>
<td>Per record</td>
</tr>
</tbody>
</table>

Note for *: recommend configuration to use PBA and PRB
Summary

New Way to Manage Data Flows
- Migrate the less accessed big data (cold) into the DCPMM and keep frequently accessed small data (hot) into the DRAM.
- With the data migration, the performance with DCPMM keep >85% performance with DRAM.

Architected for Persistence, Optimized for Performance
- Data persistent by Intel PMDK library which is performance optimized.
- Leverage the persistent capability to improve the persistent performance, redis with DCPMM persistent performance >2x open source persistent performance.
ADDITIONAL RESOURCE
Developer Call to Action

Who: Developers interested in persistent memory programming. Focus areas include: databases, large datasets, transactional programs, devops, and many others.

- **Getting Started**
  - Persistent Memory Programming - [http://pmem.io](http://pmem.io)

- **Linux* Resources**
  - Linux Community Pmem Wiki - [https://nvdimm.wiki.kernel.org/](https://nvdimm.wiki.kernel.org/)
  - Pmem enabling in SUSE Linux Enterprise 12 SP2 - [https://www.suse.com/communities/blog/nvdimm-enabling-suse-linux-enterprise-12-service-pack-2/](https://www.suse.com/communities/blog/nvdimm-enabling-suse-linux-enterprise-12-service-pack-2/)

- **Other Resources**
  - SNIA Persistent Memory Summit 2017 - [https://www.snia.org/pm-summit](https://www.snia.org/pm-summit)
  - Intel manageability tools for Pmem - [https://01.org/ixpdimm-sw/](https://01.org/ixpdimm-sw/)
  - Persistent Memory Development Kit - [https://github.com/pmem/pmdk](https://github.com/pmem/pmdk)
  - PMDK documents – [https://pmem.io](https://pmem.io)

- **Redis**
  - Redis Documents - [https://redis.io/topics/persistence](https://redis.io/topics/persistence)
  - PMEM-Redis - [https://github.com/pmem/pmem-redis](https://github.com/pmem/pmem-redis) (user mode COW + pointer based AOF)
  - Redis with PMDK POC - [https://github.com/pmem/redis/tree/reserve_publish_poc](https://github.com/pmem/redis/tree/reserve_publish_poc) + (some fix patches)

- **Rocksdb**
  - [https://github.com/pmem/pmem-rocksdb](https://github.com/pmem/pmem-rocksdb)
Provisioning Persistent Memory

- PMEM (Socket0)
  - DAX Filesystem
  - /dev/pmem0
  - Namespace0.0
  - Region 0

- PMEM (Socket1)
  - DAX Filesystem
  - /dev/pmem1
  - Namespace1.0
  - Region 1

- Persistent Memory Devices
- Namespaces
- Regions (Interleave Sets)

- DAX Filesystems: EXT4, XFS, NTFS
- Mount DAX-enabled file system onto DCPMMs

- NVDIMM Driver (Kernel)
  - /dev/pmem0
  - /dev/dax0.1
  - /dev/pmem1
  - /dev/dax1.1

- Hardware
  - Namespace0.0
  - Namespace0.1
  - Namespace1.0
  - Namespace1.1

- Volatile DRAM
  - Region 0
  - Region 1

- ndctl (Linux): configures & manages namespaces
- ipmct (Linux or UEFI): configures & manages the regions on the DIMMs

Operating System defined software device representing the namespace
Redis – Test System Setup

Server

<table>
<thead>
<tr>
<th>Component</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>SKX-2S</td>
</tr>
<tr>
<td>CPU</td>
<td>SKX</td>
</tr>
<tr>
<td>CPU per Node</td>
<td>28 core/socket, 2 sockets, 2 threads per core</td>
</tr>
<tr>
<td>Memory</td>
<td>1. 12x16GB DDR@2666 + 8x128GB AEP, QS (AD 2-2-1) or 2. 24 * 32GB DDR@2666 (Baseline)</td>
</tr>
<tr>
<td>SUT OS</td>
<td>4.17.5-100.fc27.x86_64</td>
</tr>
<tr>
<td>BKC</td>
<td>WW30.5</td>
</tr>
<tr>
<td>Network</td>
<td>80Gb Ethernet</td>
</tr>
<tr>
<td>Redis Server</td>
<td>1. Open source redis 4.0.9 for DRAM</td>
</tr>
<tr>
<td></td>
<td>2. CCE redis 4.0.9 for AD 2-2-1</td>
</tr>
<tr>
<td>Patch</td>
<td>Security patch enabled</td>
</tr>
</tbody>
</table>

Client

<table>
<thead>
<tr>
<th>Component</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>SKX-2S</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz</td>
</tr>
<tr>
<td>CPU per Node</td>
<td>24 core/socket, 2 sockets, 2 threads per core</td>
</tr>
<tr>
<td>Memory</td>
<td>196GB (12*16GB)</td>
</tr>
<tr>
<td>OS</td>
<td>4.17.5-100.fc27.x86_64 (security patch enable)</td>
</tr>
<tr>
<td>Redis Benchmark</td>
<td>Redis 4.0.9</td>
</tr>
</tbody>
</table>
Benchmark Configure*

- set,get
  
  redis-benchmark -k 1 $hostcmd -p $port -r $range -n $req_num -t set,get -d $data_size -P 3 -c 5

- set2get8 (set:get=1:4)

  memtier_benchmark $hostcmd -p $port --ratio=1:4 -d $data_size -n $req_num --key-pattern=R:R --key-minimum=1 --key-maximum=$range --threads=1 --pipeline=64 -c 3 --hide-histogram
  - bind socket "numactl –m socket taskset –c localcores"
  - range=0.1m << request 10m
  - pipeline (-P 3 for set,get ; --pipeline=64 for set2get8)
  - Client (-c 5 for set,get; -c 3 for set2get8)
  - Data size (512, 1K, 2K

*Note*: the benchmark configure parameters are the same with “1LM vs 2LM study”
Redis Performance
100% Write

- AD TPS > 90% 1LM TPS @512 within SLA (136 instances)
- AD TPS > 85% 1LM TPS @1024 data size within SLA (128 instances)
- AD TPS> 83% 1LM TPS @ 2048 data size within SLA
  - 1LM SLA instances is 92.
  - AD SLA instances is 120.
Redis Performance
100% Read

- AD QPS ~100% 1LM QPS within SLA (128 instances)
Redis Performance
SET:GET=1:4

- AD TPS > 75% 1LM TPS @512 data size
- AD TPS > 75% 1LM TPS @1024 data size
- AD TPS > 90% 1LM TPS @2048 data size

*Note*: Latency in Memtier_benchmark is average latency = total_lat/total_ops
Libpmemobj introduction

- Transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming.
- PMEMoid is like the (void *) and pointer to a object.
- Libpmemobj Actions API allows application to first reserve some persistent memory buffer in volatile state, and publish it some time later.
- libpmemobj provides ACID (Atomicity, Consistency, Isolation, Durability) transactions for persistent memory.
  - Redo log for the transaction allocation
  - Undo log for the transaction updates.

Code example:
```c
TX_BEGIN_PARAM(pool, TX_PARAM_MUTEX, &root->lock, TX_PARAM_NONE) {
    pmemobj_tx_add_range_direct(root, sizeof(*root));
    root->objA = pmemobj_tx_alloc(sizeof(struct objectA), type_num);
    pmemobj_tx_free(root->objB);
    root->objB = OID_NULL;
} TX_END
```
## Configuration

### System
- **AEP 2-2-1 and DRAM only**

### CPU
- **CLX-2S**
- **CLX 80 CPU**

### CPU per Node
- 28 core/socket, 2 sockets, 2 threads per core

### Memory
1. 12x16GB DDR@2666 + 8x128GB AEP, QS (AD 2-2-1) or
2. 24x32GB DDR@2666 (Baseline)

### SUT OS
- **Fedora**

### BKC
- WW42 BKC [AEP FW: 5310] [BIOS: PLYXCRB1.86B.0563.D11.1812050623]

### Redis Server
1. Open source redis 4.10 for DRAM
2. PMEM-redis 4.10 for AD 2-2-1; 
   - [https://github.com/pmem/pmem-redis](https://github.com/pmem/pmem-redis)

### Patch
- Security patch enabled

---

### Server (Note AOF in NVMe SSD and HD have different performance data)

<table>
<thead>
<tr>
<th>Test Command</th>
<th>benchmark</th>
</tr>
</thead>
</table>
| `numactl -m 0 taskset -c $core redis-server --port $(port) --dir ./ --appendonly yes --appendfsync always --pointer-based-aof yes --appendfilename $(port).aof --maxmemory 3G to Use the NVM: --nvm-maxcapacity 20 --nvm-dir /mnt/pmem0/ --nvm-threshold 64` | `-t: set get 
- n 1000000 
- r 12000000 
- d 1024` |

---

Start 28 redis server instances on the socket0
Start 28 redis benchmark on the socket 1

---

Redis-server1
Redis-server2
Redis-server3
Redis-server...

TCP/IP

Socket 0

Socket 1

Redis-Benchmark1
Redis-Benchmark2
Redis-Benchmark3
Redis-Benchmark...
Redis Performance (AOF)  
**pba vs Open source**

**Test case**  
Datasize: 1024, Workload: Set aof dir: nvme ssd

<table>
<thead>
<tr>
<th></th>
<th>pointer-based-aof</th>
<th>persistent ring buffer</th>
<th>appendfsy nc</th>
<th>Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td>pba-volatile</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>N/A</td>
</tr>
<tr>
<td>pba-everysec</td>
<td>yes</td>
<td>yes</td>
<td>everysec</td>
<td>Per record</td>
</tr>
<tr>
<td>pba-always</td>
<td>yes</td>
<td>yes</td>
<td>always</td>
<td>Per record</td>
</tr>
<tr>
<td>source-volatile</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>N/A</td>
</tr>
<tr>
<td>source-everysec</td>
<td>yes</td>
<td>yes</td>
<td>everysec</td>
<td>Per second</td>
</tr>
<tr>
<td>source-always</td>
<td>yes</td>
<td>yes</td>
<td>always</td>
<td>Per record</td>
</tr>
</tbody>
</table>

Note for *: recommend configuration to use PBA and PRB
## Redis Persistent Performance

<table>
<thead>
<tr>
<th>Point based AOF (PBA)</th>
<th>NO</th>
<th>YES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent ring buffer (PRB)</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Every second (ops)</td>
<td>1.74M</td>
<td>1.98M</td>
</tr>
<tr>
<td>Always (ops)</td>
<td>0.62M</td>
<td>1.41M</td>
</tr>
</tbody>
</table>

### Benefits:

- **Pointer based AOF - PBA** save value pointer on SSD instead of the whole value which reduce SSD IO throughput and solve the SSD bottleneck issue.
  - W/ PBA (every second) ~ 1.13x w/o PBA (every second)
  - W/ PBA (always) ~ 2.27x w/o PBA (always)

- **Persistent ring buffer - PRB** can keep buffer data persistent and keep durability as “always”.
  - W/ PRB every second can keep durability as “always” ~ 1.4x w/o PRB always

- To keep no data loss (PBA + PRB) performance [1.98M] ~ 3.19x Open source Always [0.62M]
Outlook of DCPMM

• PMDK is the main library for DCPMM Persistence for long time.
• eADR reduce the complexity of PMDK programming.
• In the short period, complex memory data structure can’t persistent/store to the PMEM from the performance consideration.
• New compact data structures designed for DCPMM.
• Designed K-V pair tables with PMDK for Persistence instead of AOF or WAL.
• DCPMM as SSD cache or DCPMM as the faster storage.
• Data encoding (serialization) and persistent to DCPMM (redis over flash).
• High availability design (reduce restore time) with PMDK.