ACCELERATING REDIS WITH INTEL® OPTANE™ DC PERSISTENT MEMORY

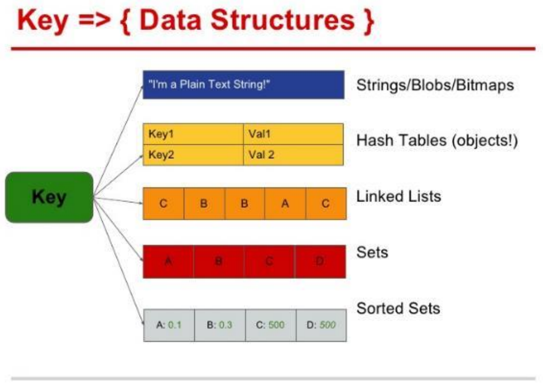
Agenda

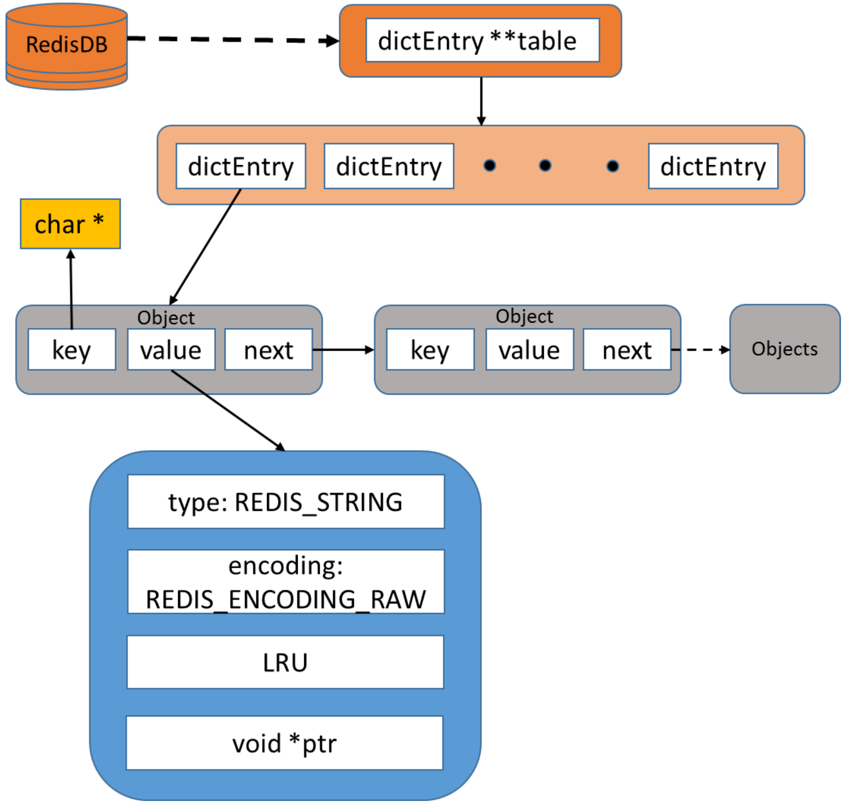
- Redis Overview
- Intel® Optane™ DC Persistent Memory (DCPMM) Introduction
  - DCPMM Operational Model
  - SNIA program model and PMDK library.
- Redis Volatile- Migrate Data to DCPMM
  - Redis Data Migration Strategy
  - Redis Data Structure Update
- Redis Persistence on DCPMM
  - Redis RDB Snapshot
  - 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.



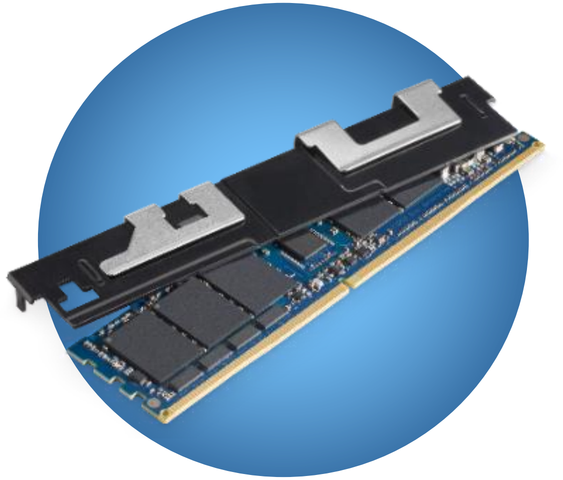


High level illustration on how content is stored in Redis

Intel® Optane™ DC Persistent Memory (DCPMM) - Operational Modes

APP DIRECT MODE

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



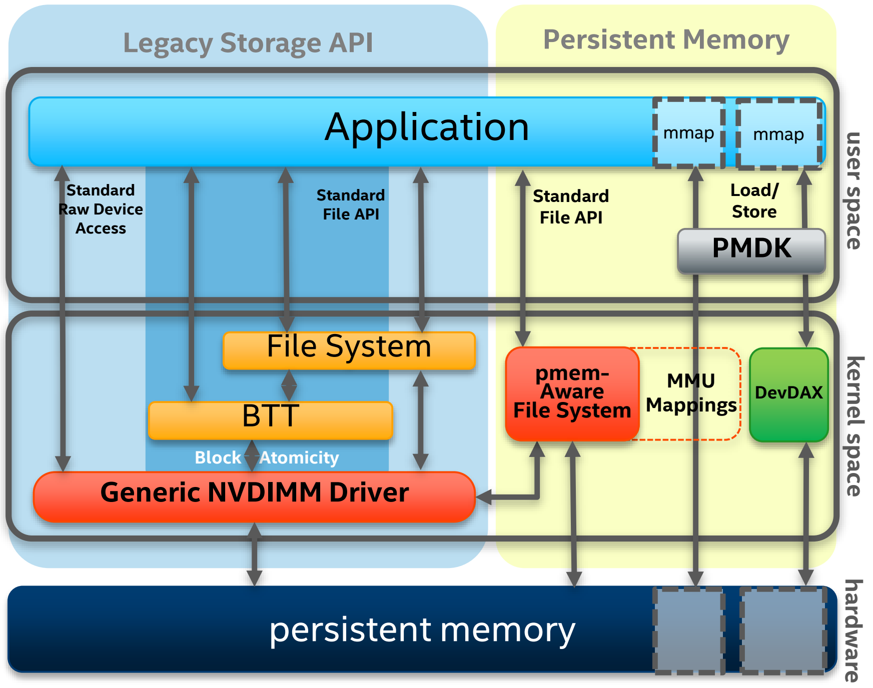
intel OPTANE™ DC PERSISTENT MEMORY

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

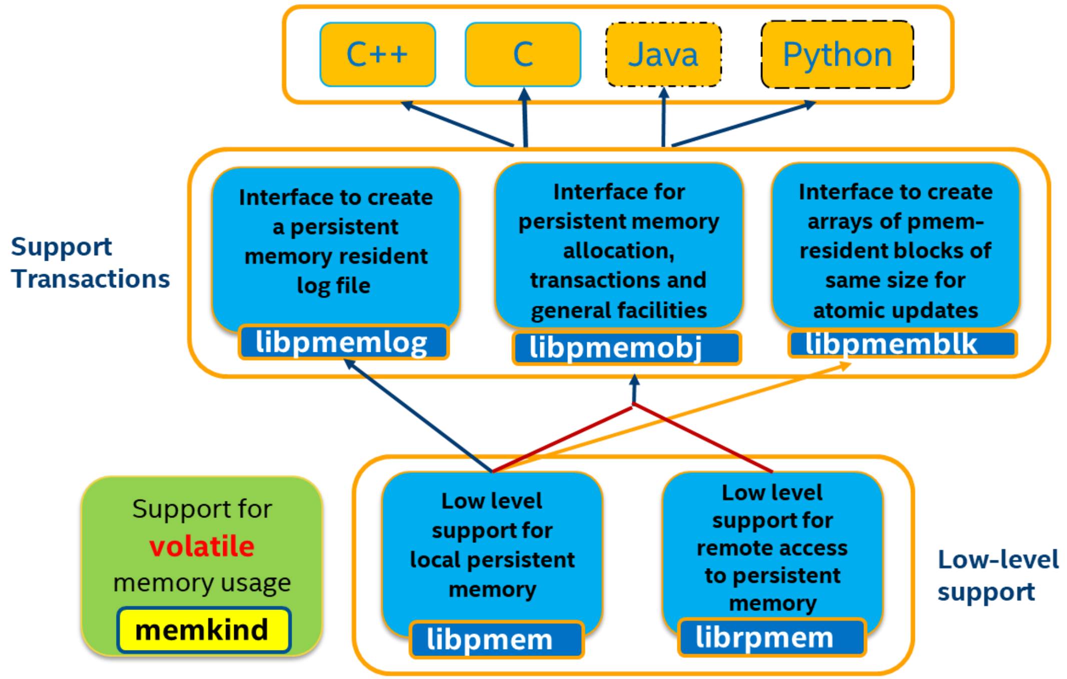
- No code changes required
- Operates in blocks like SSD/HDD
- Block atomicity

Persistent Memory

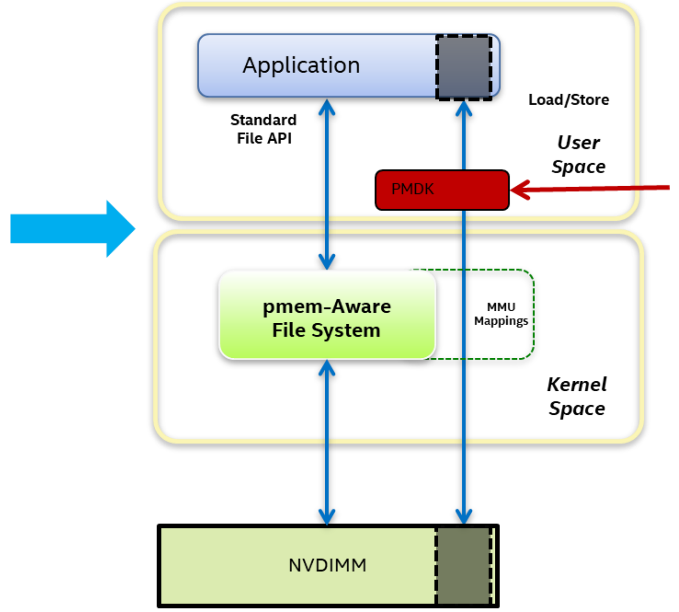
- Code changes is required, byte-addressable
- Bypasses FS page cache
- Requires DAX enabled file system "XFS, EXT4, NTFS"
- Fastest I/O path possible

App Direct "Persistent Memory" is the fastest I/O path

Persistent Library PMDK Introduction


C++
C
Java
Python
Support Transactions
Interface to create a persistent memory resident log file
libpmemlog
Interface for persistent memory allocation, transactions and general facilities
libpmemobj
Interface to create arrays of pmem-resident blocks of same size for atomic updates
libpmemblk
Support for volatile memory usage
memkind
Low level support for local persistent memory
libpmem
Low level support for remote access to persistent memory
librpmem
Low-level support

Link to Open Source : http://pmem.io/PMDK/



Link to Intel Developer Zone: https://software.intel.com/en-us/persistent-memory

PMDK is a series of persistent libraries for applications

PMDK Usage Scenarios

PMDK Libraries	Description	Use When	Persistence Need Attention
Libpmem/librpmem	Low-level library with primitives persistent APIs	Only need low-level optimized primitives API (memcpy, etc.)	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!"
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.

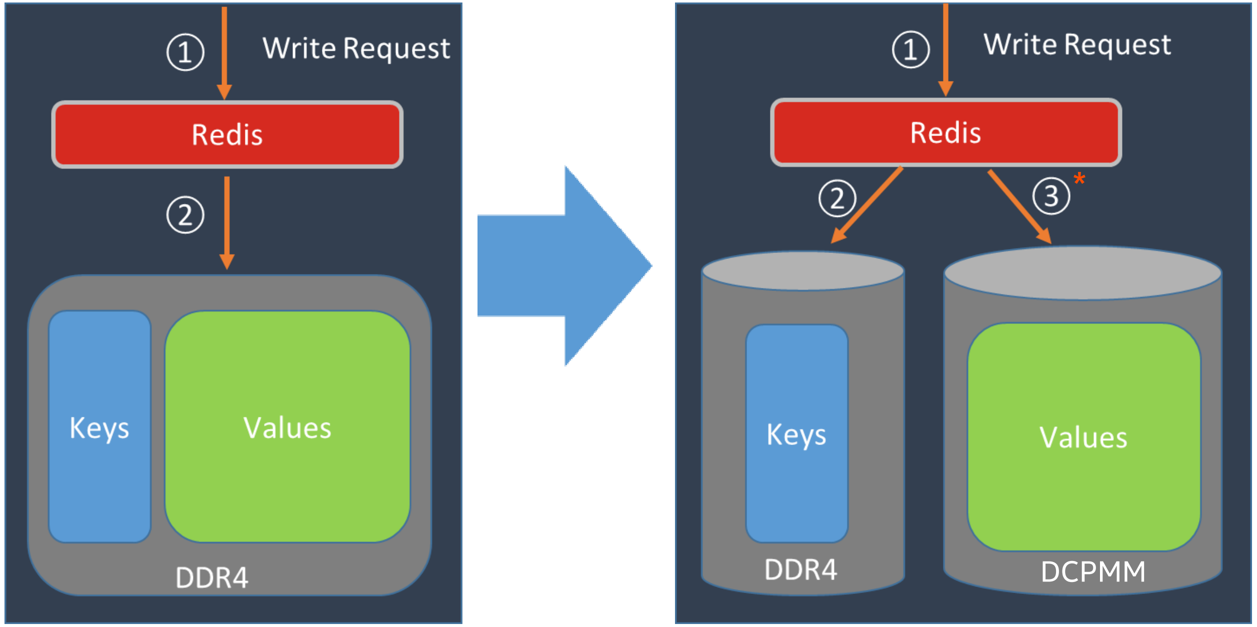
1. Application can use libmemkind+libpmem and consider data consistence and recovery mechanism from application.
2. Application can use libpmemobj and focus on optimize persistent performance.

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

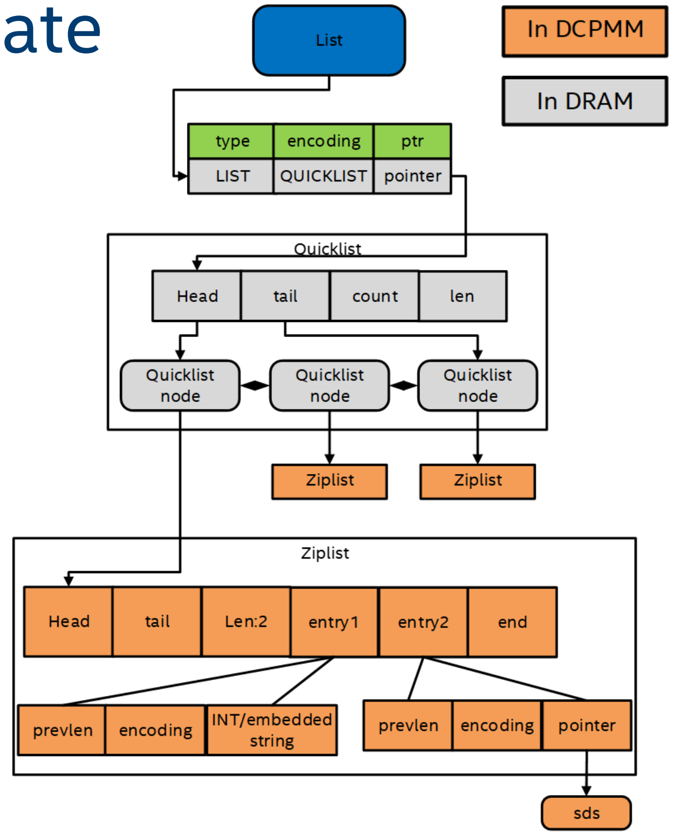
Redis Data migration strategy



Note *: value >= threshold will move to DCPMM, Keys keep in DRAM (metadata: data structures to manage key-value keep in DRAM)

Redis typical data structures Update

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

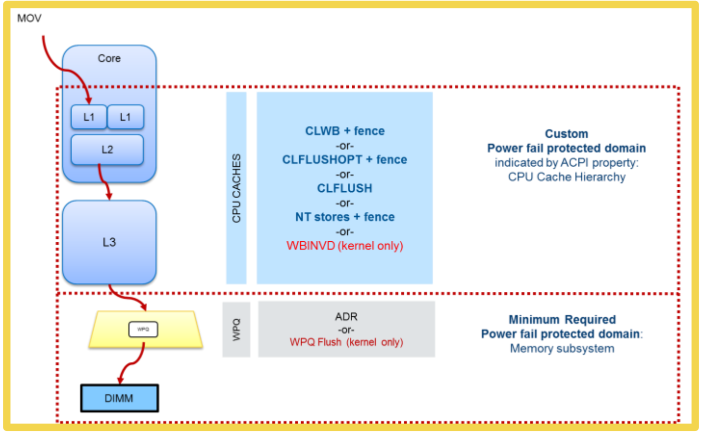


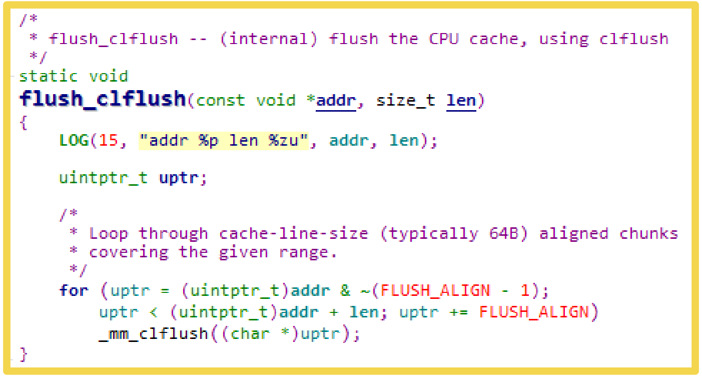
Note: With 2-2-1, 40*2 Gbps network, 1k data and SLA<99% latency 1ms; Redis volatile performance with DCPMM is quite close to the DRAM with redis-benchmark. 100% write > 85% performance and 100% read ~100% performance of opensource with DRAM.

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: Intenal Tuning guide with BKMs in bi-weekly PnP report. (link: CLX_AEP PnP Tuning Guide)*



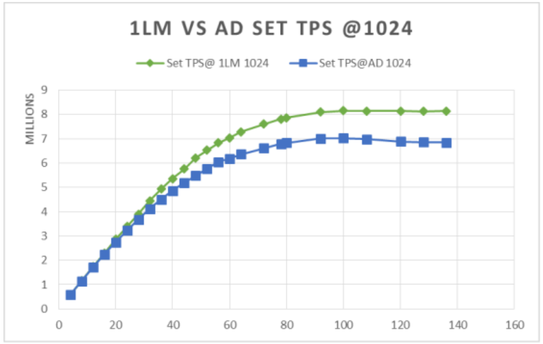


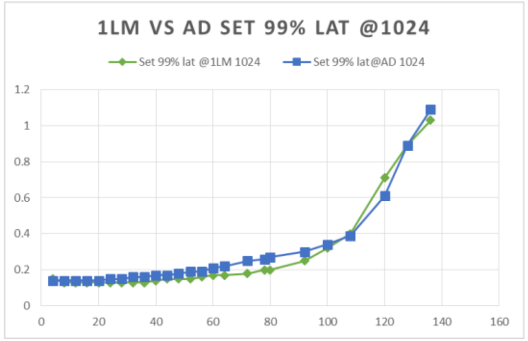
```
/*
 * flush_clflush -- (internal) flush the CPU cache, using clflush
 */
static void
flush_clflush(const void *addr, size_t len)
{
	LOG(15, "addr %p len %zu", addr, len);

	uintptr_t uptr;

	/*
	 * Loop through cache-line-size (typically 64B) aligned chunks
	 * covering the given range.
	 */
	for (uptr = (uintptr_t)addr & ~(FLUSH_ALIGN - 1);
		uptr < (uintptr_t)addr + len; uptr += FLUSH_ALIGN)
		_mm_clflush((char *)uptr);
}
```

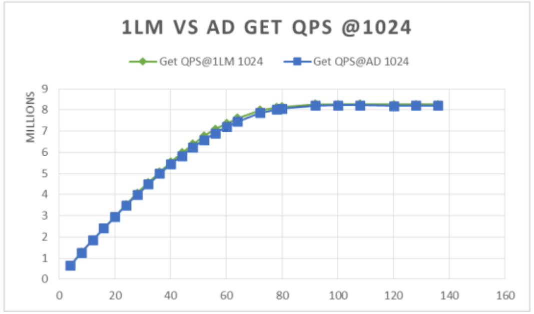
Redis Volatile Performance with DCPMM

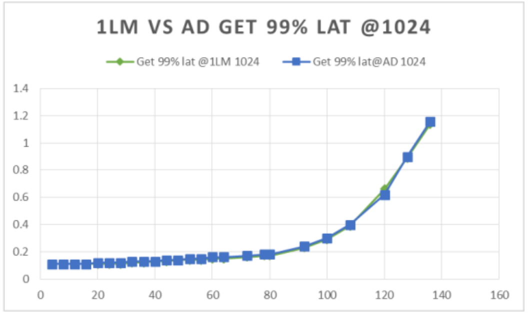
- Redis with DCPMM write TPS > 85% DRAM TPS @1024 data size within SLA (128 instances)
- Redis with DCPMM read QPS ~100% DRAM QPS @1024 data size within SLA

Note * SLA is 99% latency <1ms



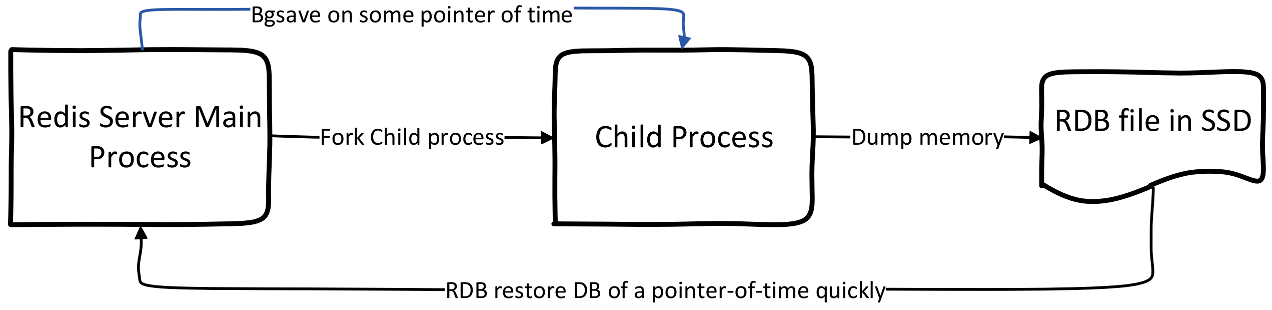




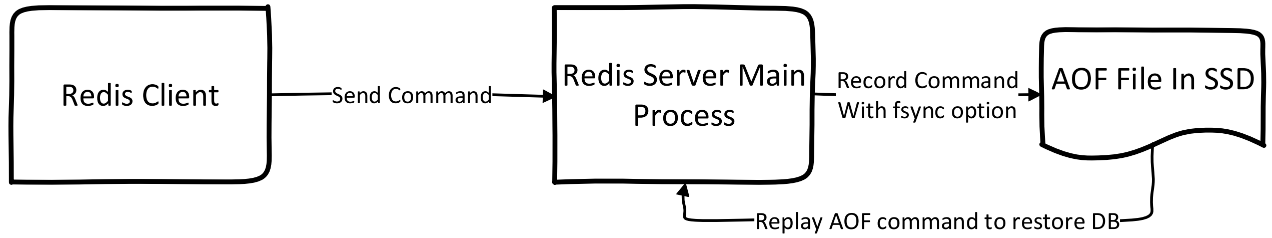


REDIS PERSISTENCY ON DCPMM

Redis Persistence Introduction


Bgsave on some pointer of time
Redis Server Main Process
Fork Child process
Child Process
Dump memory
RDB file in SSD
RDB restore DB of a pointer-of-time quickly

Redis Persistence: RDB used for the disaster recovery


Redis Client
Send Command
Redis Server Main Process
Record Command With fsync option
AOF File In SSD
Replay AOF command to restore DB

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.

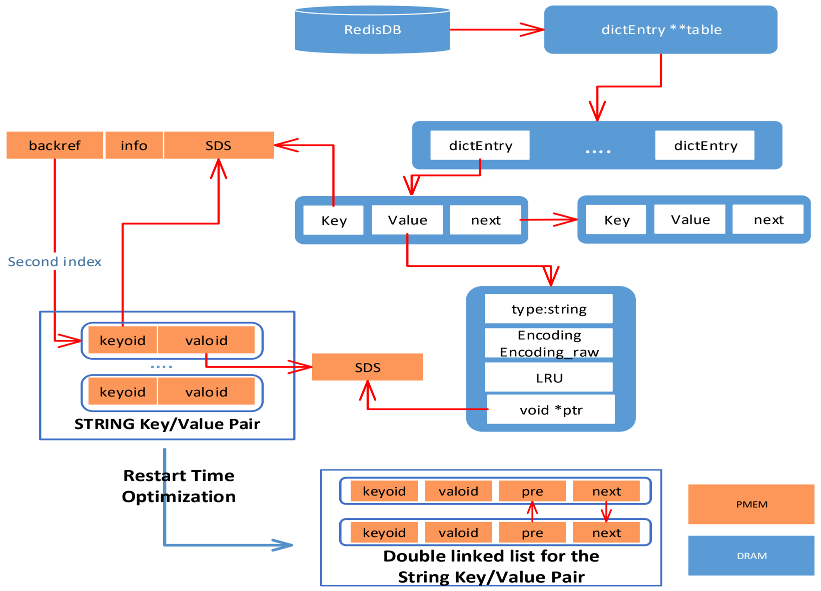


Fig1. Redis string persistent with libpmemobj

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.

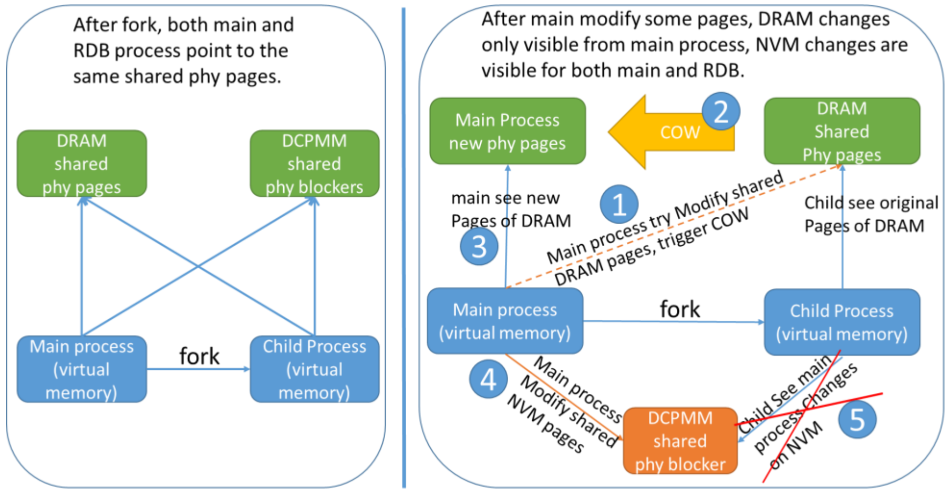

After fork, both main and RDB process point to the same shared phy pages.
DRAM shared phy pages
DCPMM shared phy blockers
Main process (virtual memory)
fork
Child Process (virtual memory)
After main modify some pages, DRAM changes only visible from main process, NVM changes are visible for both main and RDB.
Main Process new phy pages
COW
2
DRAM Shared Phy pages
main see new Pages of DRAM
3
1
Main process try Modify shared DRAM pages, trigger COW
Child see original Pages of DRAM
Main process (virtual memory)
fork
Child Process (virtual memory)
4
Main process Modify shared NVM pages
DCPMM shared phy blocker
~~Child See main process changes on NVM~~
5

Fig 1. DAX-enabled DCPMM doesn't support the COW

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.

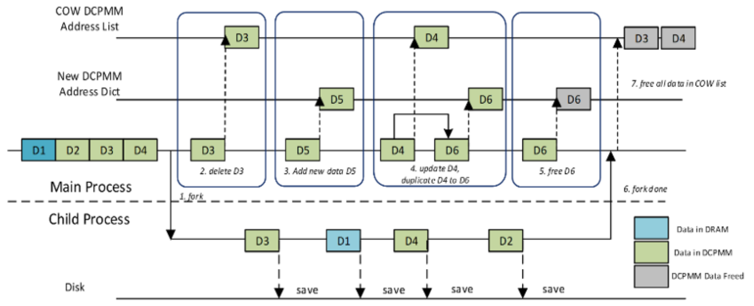


Figure 1 User COW implementation in detail

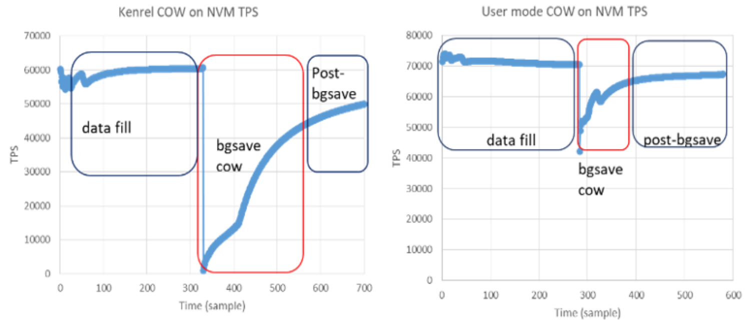
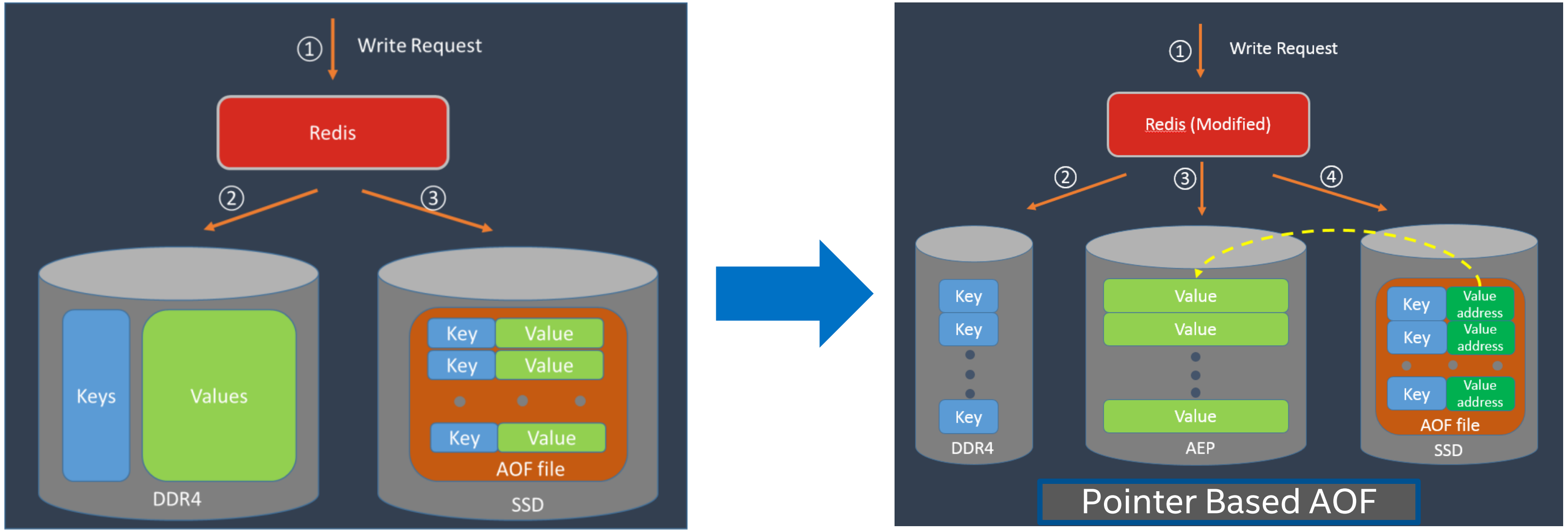


Figure 2 User COW shows good performance ~3.x during snapshot

User Mode COW designed for RDB with DCPMM and show ~3x write performance in some stress scenarios.

Redis Pointer Based AOF



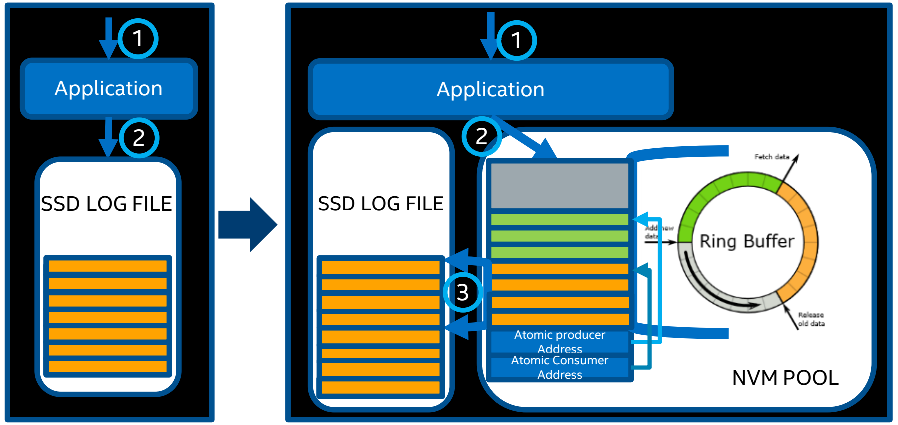
1. Redis get a write request from client
2. Redis store the key in the DRAM
3. Redis use memkind to allocate space and store the big value data into PMEM and use libpmem persistent the big value.
4. Redis save the command as AOF in SSD. AOF only store the key and location of big value(DCPMM)

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 perfomrnace. 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.

Persistent Ring Buffer used for SSD log acceleration

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.*

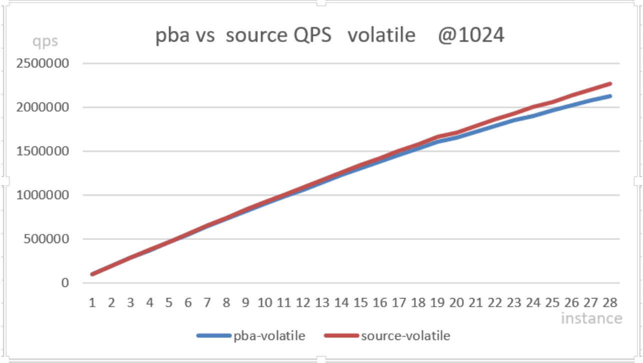
Redis Performance (AOF) pba vs Open source

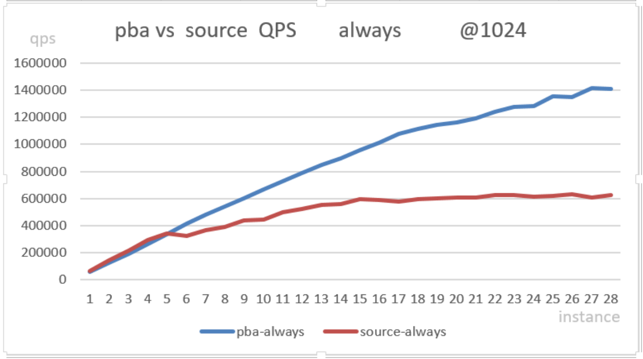
Test case

datasize: 1024,
Workload: Set
aof dir : nvme ssd

	pointer-based-aof	persistent ring buffer	appendfsync	Durability
pba	yes	yes	Everysec	Per record
* Open source-volatile	no	no	no	N/A
source-always	yes	yes	always	Per record

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

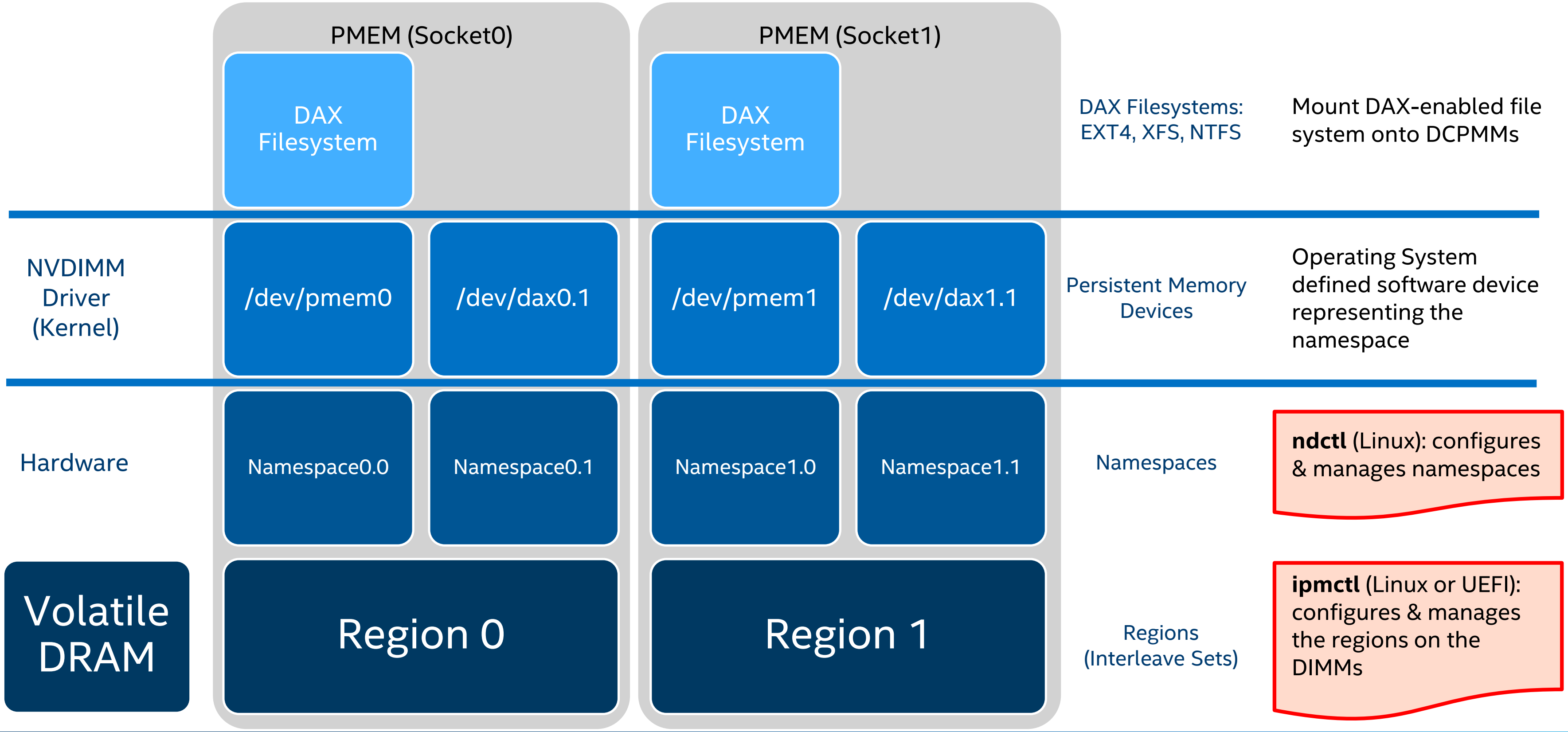
ADDITIONAL RESOURCE

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
  - Intel IDZ Persistent Memory- https://software.intel.com/en-us/persistent-memory
  - Persistent Memory Programming - http://pmem.io
- Linux* Resources
  - Linux Community Pmem Wiki - 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/
- Other Resources
  - SNIA Persistent Memory Summit 2017 - https://www.snia.org/pm-summit
  - Intel manageability tools for Pmem - https://01.org/ixpdimm-sw/
  - Persistent Memory Development Kit - https://github.com/pmem/pmdk
  - PMDK documents – https://pmem.io
- Redis
  - Redis Documents - https://redis.io/topics/persistence
  - 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 + (some fix patches)
- Rocksdb
  - https://github.com/pmem/pmem-rocksdb

Provisioning Persistent Memory



Mount DAX-enabled file system onto DCPMMs

Operating System defined software device representing the namespace

ndctl (Linux): configures & manages namespaces

ipmctl (Linux or UEFI): configures & manages the regions on the DIMMs

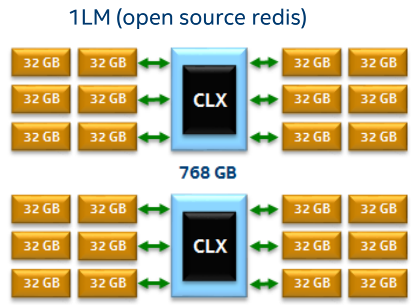
Redis – Test System Setup

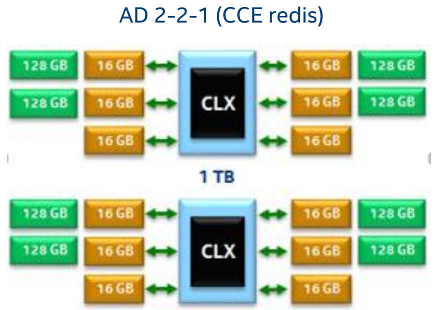
Server

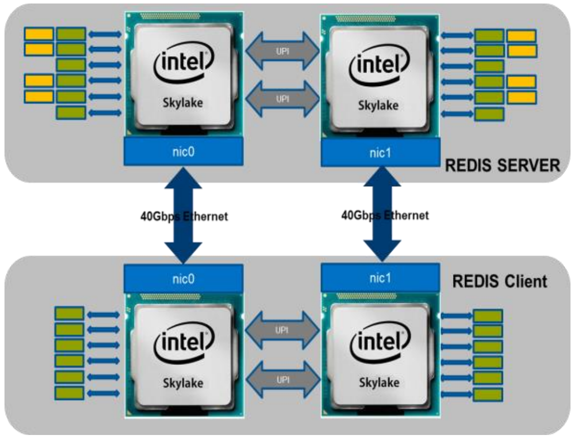
	AEP 2-2-1 and DRAM only
System	SKX-2S
CPU	SKX
CPU per Node	28core/socket, 2 sockets, 2 threads per core
Memory	1. 12x16GB DDR@2666 + 8x128GB AEP, QS (AD 2-2-1) or 2. 24 * 32GB DDR@2666 (Baseline)
SUT OS	4.17.5-100.fc27.x86_64
BKC	WW30.5
Network	80Gb Ethernet
Redis Server	1. Open source redis 4.0.9 for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

Client

Client	
System	SKX-2S
CPU	Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
CPU per Node	24core/socket, 2 sockets, 2 threads per core
Memory	196GB (12*16GB)
OS	4.17.5-100.fc27.x86_64 (security patch enable)
Redis Benchmark	Redis 4.0.9







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)

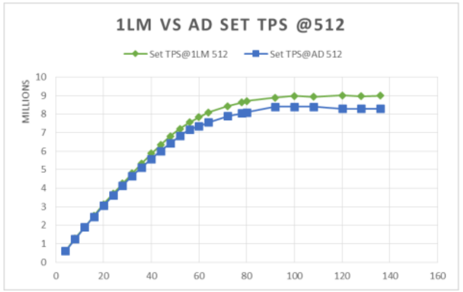
memtier_benchmark $hostcmd -p $port --ratio=1:4 -d $data_size -n $req_num --key-pattern=R:R --key-minimum=1 --key-maximum=$range --threads=1 --pipeline=64 -c 3 --hide-histogram

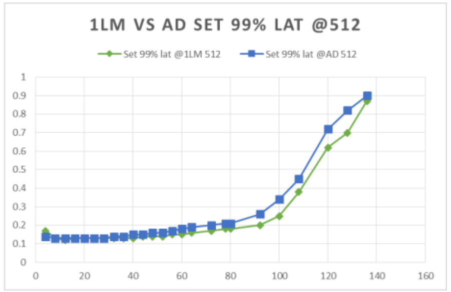
- bind socket “numactl –m socket taskset –c localcores”
- range=0.1m << request 10m
- pipeline (-P 3 for set,get ; --pipeline=64 for set2get8)
- Client (-c 5 for set,get; -c 3 for set2get8)
- Data size (512, 1K, 2K

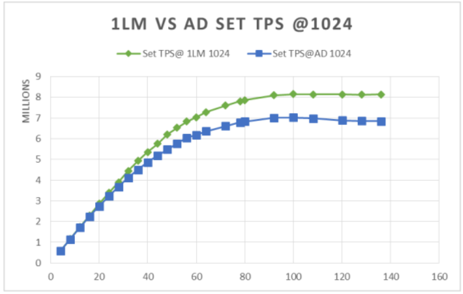
Note *: the benchmark configure parameters are the same with “1LM vs 2LM study”

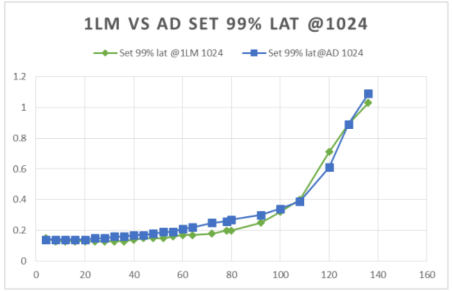
Redis Performance 100% Write

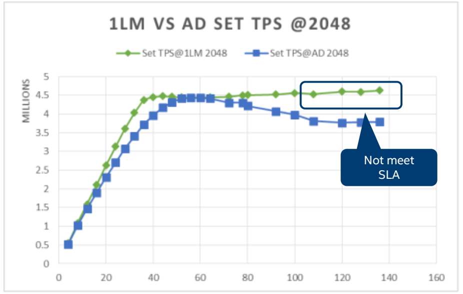
- AD TPS > 90% 1LM TPS @512 within SLA (136 instances)
- AD TPS > 85% 1LM TPS @1024 data size within SLA (128 instances)
- AD TPS> 83% 1LM TPS @ 2048 data size within SLA
  - 1LM SLA instances is 92.
  - AD SLA instances is 120.

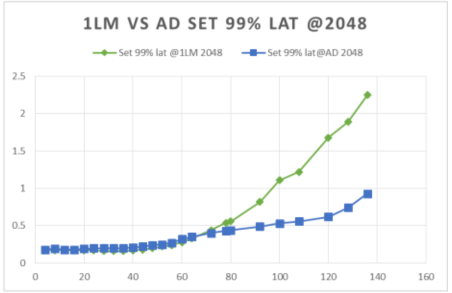






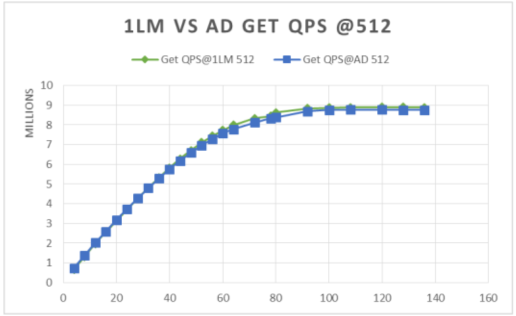


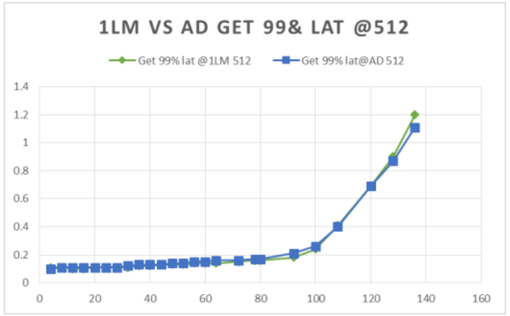


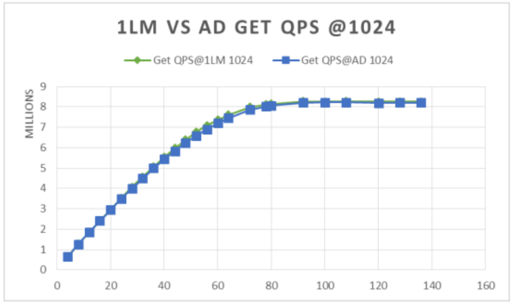


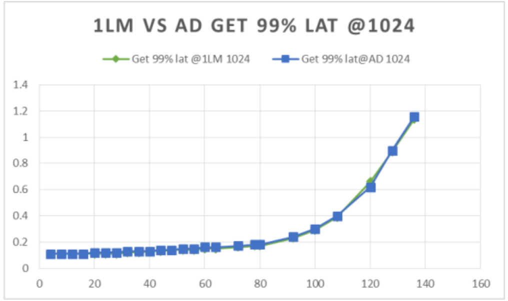
Redis Performance 100% Read

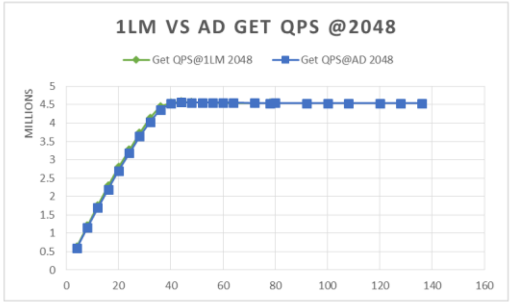
- AD QPS ~100% 1LM QPS within SLA (128 instances)

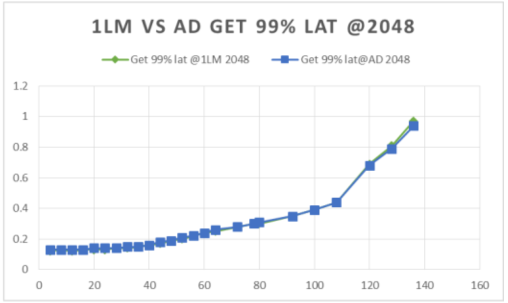








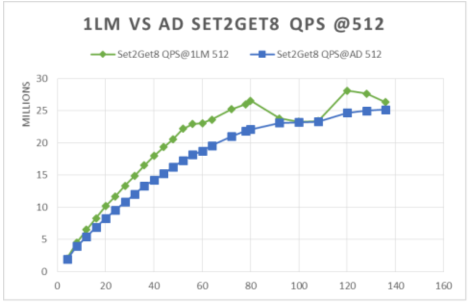


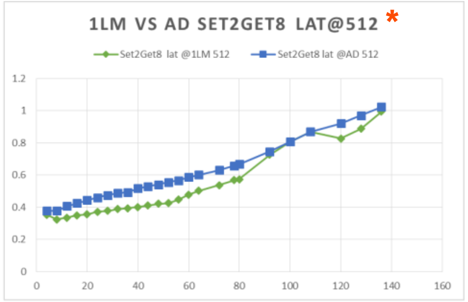


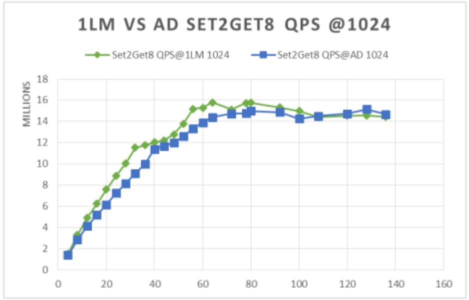
Redis Performance SET:GET=1:4

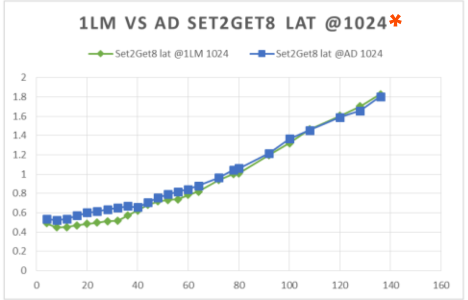
- AD TPS > 75% 1LM TPS @512 data size
- AD TPS > 75% 1LM TPS @1024 data size
- AD TPS> 90% 1LM TPS @ 2048 data size

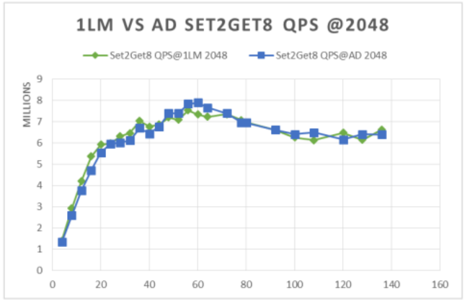
Note: Latency in Memtier_benchmark is average latency = total_lat/total_ops*

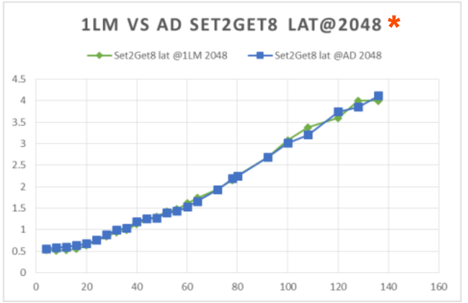






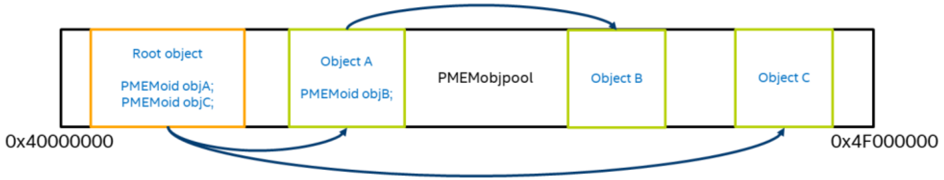






Libpmemobj introduction

- 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.



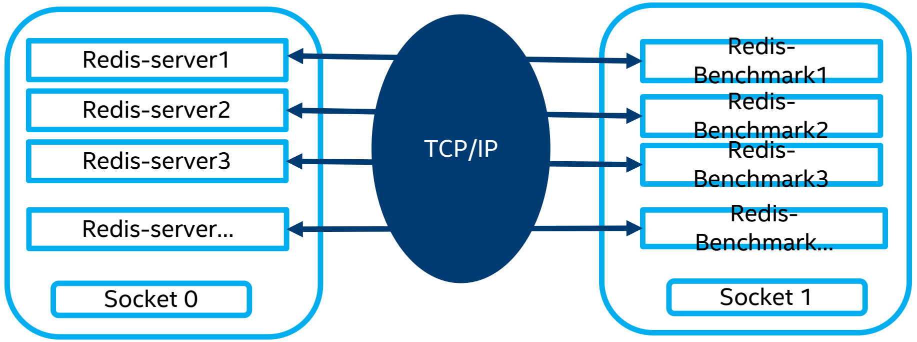
Code example: Allocate new objects / Free existing objects/ Modify existing objects/Isolate objects

```
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

	AEP 2-2-1 and DRAM only
System	CLX-2S
CPU	CLX B0 CPU
CPU per Node	28core/socket, 2 sockets, 2 threads per core
Memory	1. 12x16GB DDR@2666 + 8x128GB AEP, QS (AD 2-2-1) or 2. 24 * 32GB 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
Patch	Security patch enabled

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



	Server (Note AOF in NVMe SSD and HD have different performance data)	benchmark
Test Command	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

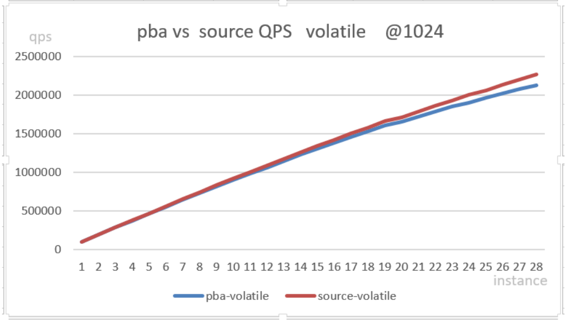
Redis Performance (AOF) pba vs Open source

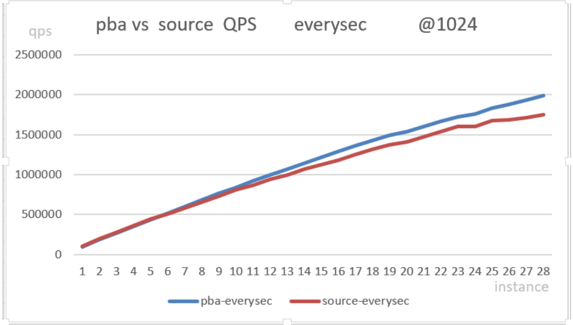
Test case

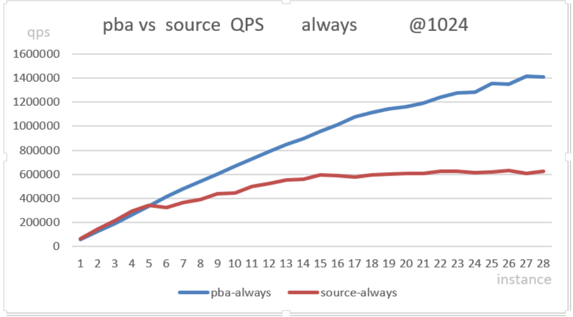
datasize: 1024,
Workload: Set
aof dir : nvme ssd

	pointer-based-aof	persistent ring buffer	appendfsync	Durability
pba-volatile	no	no	no	N/A
* pba-everysec	yes	yes	everysec	Per record
pba-always	yes	yes	always	Per record
source-volatile	no	no	no	N/A
source-everysec	yes	yes	everysec	Per second
source-always	yes	yes	always	Per record

Note for *: recommend configuration to use PBA and PRB







Redis Persistent Performance

Point based AOF (PBA)	NO	YES	
Persistent ring buffer(PRB)	**NO**	**NO**	**YES**
Every second (ops)	1.74M $\xrightarrow{113.7\%}$	1.98M	1.98M
Always (ops)	0.62M $\xrightarrow{227\%}$	1.41M $\xrightarrow{140\%}$	

Perf gain

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.