UDORN: A Design Framework of Persistent In-Memory Key-value Database for NVM

Xianzhang Chen, Edwin H.-M. Sha, Ahmad Abdullah, Qingfeng Zhuge, Lin Wu, Chaoshu Yang, and Weiwen Jiang
College of Computer Science, Chongqing University, Chongqing, China.
Email: {xzchen109, edwinsha, ahmadin4matic, qfzhuge, wujingswust, yangchaoshu, jiang.wwen}@gmail.com

***Abstract*—Emerging non-volatile memory (NVM) technologies provide opportunities to improve the performance of key-value databases (KVDBs) by deploying database on NVM. However, existing in-memory KVDBs cannot fully exploit the advantages of NVM. They process data on in-memory database and store an image on persistent storage via an underlying file system. The performance of database operations is degraded by the backup mechanisms and involved I/O routines. In this paper, we propose a new design framework of in-memory KVDB called Unified Database on Raw NVM (UDORN). In UDORN, a persistent database on NVM is employed to accomplish the functions of both conventional in-memory database and persistent image. During runtime, the persistent database is mapped to process address space. The operations are directly performed on NVM via the corresponding address space. We implement a case study of UDORN based on open-source in-memory KVDB Redis. Compared with original Redis, UDORN achieves more than 1400 times and 84% performance improvement when Redis deploys backup image on HDD and memory, respectively. Compared with the enhanced Redis using the NVM Library, UDORN also achieves 6 times performance improvement.**

I. Introduction

Key-value databases (KVDBs) are widely employed as a data serving layer in various scenarios, such as Internet-oriented datacenters [1], mobile devices [2], [3], and graph processing platforms [4]. As the basic data store of applications, deploying KVDB in memory can benefit the data processing performance of the associated applications. Recently, emerging non-volatile memory (NVM) technologies, such as 3D Xpoint [5], Phase Change Memory (PCM) [6], STT-RAM [7], and racetrack memory [8], have become attractive storage of in-memory KVDB for their high speed, byte-addressability, and non-volatility.

Nevertheless, existing in-memory KVDBs [9], [10] cannot fully exploit the advantages of NVM for they are designed for conventional volatile memory based architectures. Conventionally, an in-memory KVDB is comprised of a temporary database in main memory (i.e., in-memory database) and a persistent image on persistent storages. On one hand, the in-memory database cannot benefit from the non-volatility of NVM since they rely on the temporary data structures managed by the kernel. Thus, the updates performed on the in-memory database need backups on the persistent image.

On the other hand, the persistent image cannot exploit the byte-addressability of NVM. A persistent image contains a set of files belonging to the underlying file system. Even though NVM is used as storage, the persistent image still need an in-memory file system [11]–[16], such as Ext4-DAX [16], to manage the storage of database. Therefore, writing a log of database can even trigger a journaling operation of the underlying file system. Existing in-memory KVDBs have large overhead for maintaining the persistent image via slow I/O routines.

Similarly, another solution deploys KVDB on NVM via the persistent memory (PMEM) programming libraries, such as the NVM Library (NVML) [17]. This approach enables KVDB to direct access NVM. However, KVDB is also limited by NVML for the programming model regards NVM as a memory-mapped file. For example, the NVML-enhanced Redis [18] (denoted by Redis-NVML) only supports string values and cannot perform delete operations.

In this paper, we use NVM as the storage of KVDB and propose a new design framework of in-memory KVDB, Unified Database on Raw NVM device (UDORN), for achieving high performance. UDORN combines the functions of in-memory database and persistent image into one "unified database" on NVM device. On one hand, UDORN maintains only one "persistent database" on NVM and expose it to the process address space during runtime. The updates are directly performed on the persistent database. On the other hand, UDORN manages the raw NVM device by the database itself rather than an underlying file system. The operations of the unified database can be performed in the manner of pure memory rather than complex I/O routines. The consistency mechanisms of database can also be simplified taking advantages of the byte-addressability of NVM. As a consequence, UDORN is expected to provide high performance by avoiding I/O layers and using simplified operations.

To evaluate the proposed design ideas, we implement a case study based on the open-source in-memory KVDB Redis [9]. Concretely, UDORN implements the metadata structures for NVM management, persistency of database, and the consistency mechanism of updating operations. In the experiments, UDORN is compared with Redis-NVML and the original Redis using HDD and memory as backup storage. We measure the launch time, turn-off time, and workloads via *redis-benchmark* [19]. Experimental results show that UDORN achieves more than 6 times, 1400 times, and 84% performance improvement over Redis-NVML, Redis using HDD, and Redis using memory, respectively. The launch speed and turn-off speed of UDORN can be thousands times faster than various configurations of Redis when the number of key-value pairs grows large.

II. Background and Design Principles

Key-value databases (KVDB) have been a research topic for decades [20]–[28]. Recently, the rapidly growing NVMs, such as 3D Xpoint [5] and STT-RAM [7], are becoming attractive candidates to replace block devices for their near-DRAM speed, byte-addressability, and high storage density. It is promising to employ NVM as the persistent storage. The design of KVDB in memory needs to be restudied. In this section, we review existing architectures of in-memory KVDB and discuss the design principles of in-memory KVDB for emerging NVM.

A. The architectures of in-memory KVDBs

In-memory database and persistent image on block storages. Most existing in-memory KVDBs [9] are based on DRAM and persistent block storages, such as HDD and SSD. An illustration of their architecture is shown in Fig. 1(a). Generally, such a KVDB consists of an in-memory database and a persistent image on block

This work is partially supported by National 863 Program 2015AA015304, Chongqing High-Tech Research Program cstc2014yykfB40007, NSFC 61472052.
978-1-5386-1768-7/17/$31.00 ©2017 IEEE

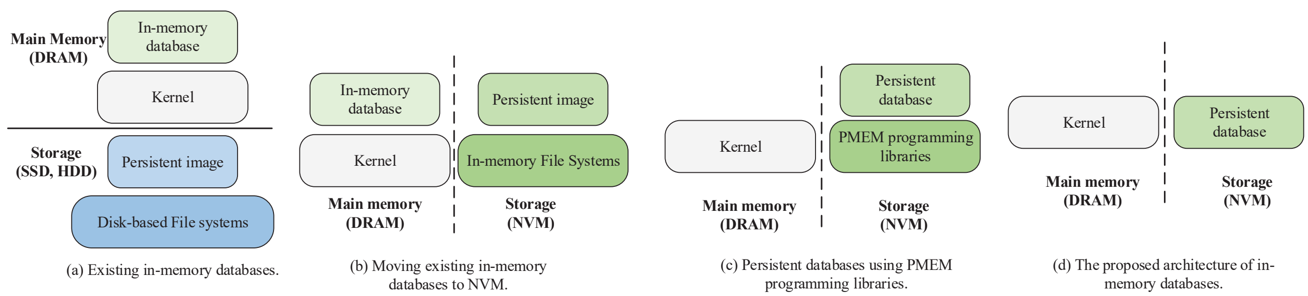


Fig. 1. Illustration of the architectures of in-memory databases.

storages. On one hand, the physical memory and the data structures of the in-memory database are fully managed by the kernel. Each time an in-memory database is created, it calls the operating system to allocate new physical memory and build new data structures. Hence, the in-memory database is temporary and cannot survive system reboot. They need to back up the update operations persistently on storage.

On the other hand, the persistent image stored on secondary storage is built upon disk-based file systems, such as EXT4. Therefore, to create the database in memory, the system needs to load data from the files via slow block I/O operations. Similarly, the system also has to go through block I/O routines in backing up the updates to storage. In consequence, existing designs have large overhead in establishing the in-memory database and maintaining the persistent image.

In-memory database and persistent image on NVM. While employing NVM as storage, the direct approach is to store the persistent image of KVDB on NVM via persistent in-memory file systems [11]–[16]. An illustration of the architecture is shown in Fig. 1(b). In this scenario, NVM is managed by underlying persistent in-memory file systems, such as EXT4 with Direct Access (DAX) [16] support. Then, KVDB can benefit from the fast in-memory data accesses and save the cost for traversing the slow traditional block I/O routines.

Nevertheless, the architecture in Fig. 1(b) still has large overhead:

1) The process of the database still constructs a temporary in-memory database in DRAM. The updates to the in-memory database should be backed up to the persistent image.
2) The performance and persistency of the database rely on the efficiency of the underlying file system. The software routines of file I/Os, such as the Virtual File System (VFS) layer, exhibits large costs.

Therefore, simply deploying the persistent image of database on in-memory file systems cannot fully exploit the advantages of NVM-based storages.

Persistent database based on NVM library. A key advantage of NVM is that processes can directly access NVM via virtual address spaces rather than I/O operations. To exploit this advantage, a typical approach is to deploy KVDB on NVM via the persistent memory (PMEM) programming libraries [17], as shown in Fig. 1(c). In this architecture, the KVDB is a persistent database on NVM. The programming library is responsible for managing NVM. The PMEM programming model for NVM is based on memory-mapped files.

For example, the widely used in-memory KVDB Redis [9] has been extended to a persistent database on NVM (denoted by Redis-PMEM) via the PMEM programming library "NVM Library" [18]. In the PMEM programming model, a NVM device acts as a memory-mapped file in kernel. The NVM Library provides seven libraries to manage NVM, such as the *libpmemlog* library that provides a log file on NVM. Taking advantages of the NVM Library, Redis-PMEM allows user to access NVM via direct load/store instructions and improves the logging mechanisms. When running Redis-PMEM in Append Only File (AOF) mode, all the commands can be saved in a pmem-resident log file, instead of a plain-text append-only file stored on a conventional hard disk drive.

However, a KVDB using this architecture is limited by PMEM programming model for NVM is managed by the NVM Library. Concretely, there are two limitations:

1) The NVM library brings overhead for managing the NVM space as files, such as the log file of *libpmemlog*.
2) It is difficult to support the pointer-based data types for the whole KVDB is placed in a memory-mapped file. For example, Redis-PMEM only supports strings.

B. *Design principles*

To exploit the benefits of NVM, we propose a new architecture of in-memory KVDB for NVM, as shown in Fig. 1(d). In the architecture, the in-memory database and persistent image are combined to one "persistent database". The persistent database builds on NVM device directly. The database is mapped to process virtual address space and accessed by load/store instructions rather than slow file I/Os. The KVDB manages NVM by itself other than the NVM libraries. The consistency of operations and the management of NVM can be improved to achieve high performance. In designing such an in-memory KVDB for NVM, we have three principles:

1) The database should be persistently stored on NVM so that it can be recovered from the process exit or system reboot.
2) The space of the database cannot be swapped out by the virtual memory management of the kernel.
3) The management of NVM should be independent from the memory management of the kernel, so as not to reclaims the physical memory of KVDB during system reboot.

III. Proposed Design Framework of Persistent In-Memory KVDB for NVM

In this section, we present a design of persistent in-memory database for NVM, namely, Unified Database On Raw NVM (U-DORN). In the aim of achieving high performance, the database under the proposed design is persistently stored on NVM without using a file system or PMEM programming library. To ensure the persistency and consistency of the database, we propose the design of persistent metadata, management of NVM, and the optimized consistent operations.

A. *An Overview of Unified Database on Raw NVM*

The proposed UDORN is based on the concept that a persistent in-memory KVDB is "a unifier of in-memory database and persistent image on raw NVM". A persistent database on NVM acts as an in-memory database during runtime and functions as a persistent image.

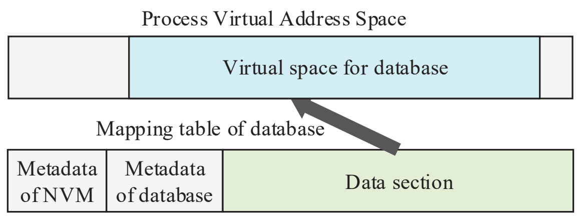


Fig. 2. Layout of physical NVM space.

On one hand, the data of the database are persistently stored on raw NVM device utilizing the non-volatility of NVM. Actually, disk-based database designs have tried to store database directly on raw devices, such as Oracle 11g [29]. However, the high costs for creating and managing raw disk-based devices counteract the benefits of bypassing file system layers. On the contrary, raw NVM devices can be used as direct storage of database because of their advanced characteristics:

1) NVM can be connected to memory bus rather than I/O bus. Thus, the difficulty of software routines for accessing and managing raw block devices, such as block driver, are irrespective in the management of raw NVM devices.
2) NVM device is byte-addressable and random accessible via `load/store` instructions. As a result, the organizing data structures of the database can be maintained by virtual address space. Hence, the file system layers is not necessary for a database on raw NVM device.

When the database is deployed on raw NVM device, the database itself needs to manage the NVM device, such as the device information, the layout of data sections on NVM, and the free NVM space.

On the other hand, a persistent database stored on NVM can be exposed to process virtual address space taking advantages of the byte-addressability of NVM, as shown in Fig. 2. Once the persistent database is exposed to process virtual address space, it behaves similarly to in-memory database. Concretely, the persistent database can be located by the corresponding process virtual address. The data are efficiently accessed via the virtual addresses and the address translation hardware MMU, rather than going though the slow file I/O routines.

Different from conventional in-memory database, the persistent database can survive system reboots. To make the database persistent on NVM, we propose that the database needs not only a dedicated physical memory management mechanism of NVM, but also new metadata of database.

B. Metadata for Persistency

As shown in Fig. 2, the physical NVM in UDORN is partitioned into three sections: metadata of NVM, metadata of database, and data section. The metadata of NVM and the metadata of database are bound together to ensure data persistency.

1) Metadata of NVM: In UDORN, an in-memory KVDB manages two kinds of information of NVM device: the information of NVM and NVM space management. The information of NVM are the metadata related to NVM and database storage, such as the physical size of NVM, the size of the metadata of database, and the pointer to the metadata of the database. These information are maintained in a fixed physical location of NVM. In Linux kernel, a NVM device is regarded as a file and has implemented a set of basic operations for the device, such as *open*(), *read*(), and *write*(). The information of NVM can be obtained by opening NVM device and reading the corresponding physical location reserved for the metadata of NVM.

On one hand, to be independent from the memory management of the kernel, such as buddy system and slab allocator, and ensure data persistency, the persistent database needs to manage the NVM space by itself. In UDORN, the persistent database allocates and reclaims physical NVM via a dedicated memory allocator. The allocator consists of a space management mechanism and a set of dedicated interfaces. On the other hand, the persistent database on NVM still reuses the memory management mechanisms in existing computer systems since there are many hardware optimizations, such as MMU and TLB, for speeding up the translation procedures of virtual address to physical address.

The design of space management need to consider the characteristics of KVDB and NVM. For example, the requested size of key-value database is often less than a 4KB. Thus, the allocator may need to support fine-grained memory allocation/deallocation. Besides, the durability and consistency of the operations for allocating or reclaiming physical space are also necessary for NVM storage.

2) Metadata of In-Memory Database: Even though the data of a persistent database is preserved in NVM, it still needs information to locate and recover the organization structures of the persistent database when launching the database. In UDORN, we propose to manage the organization structures of persistent database via a dedicated structure called "metadata of database". In order to recover the in-memory database, the metadata of database can be stored on a location next to the metadata of NVM, as shown in Fig. 2.

Basically, the metadata of database maintains four data structures: the mapping table of persistent database, the size of the database, the index of the whole database structures, and the log of database. The mapping table organizes the in-use physical memory space of the persistent database. Its structure is in the same form of the system page table. When the process of a persistent database in UDORN is initialized, the persistent database on NVM is exposed to the process virtual address space by inserting the mapping table into the page table of the process. Then, the user can access the persistent database on NVM via its process virtual address space. The size of acquired process virtual address space is equal to the maximum size of the persistent database.

Although the database is exposed to process virtual address space, the process still needs to understand the layout of the database in the virtual address space. Therefore, a virtual space management is required for allocating and reclaiming the free virtual address space belonging to database. We illustrate the metadata of database by a case study in Section IV-B.

C. Consistency of Operations

Different from existing in-memory KVDBs, the operations of UDORN are performed directly on the persistent database rather than a temporary database in main memory. Hence, any operation that modifies the metadata or data of the in-memory database should be consistent and durable on NVM. In this paper, we perform the in-memory operations as transactions to ensure the consistency and durability of database.

Conventional in-memory KVDB use *write-ahead logging* (or *journaling*) [30], [31] to ensure the consistency of operations. With write-ahead logging, an in-memory database needs to backup both the data and command of the operation before performing the operation. The backups are usually written to a log on the storage. The persistent image may also need to be updated even the in-memory database has been updated. The logs and persistent image are files belonging to the file system that manages the storage. Therefore, writing log files and updating the persistent image all implicitly go through the consistency mechanism of the underlying file system. As a consequence, an

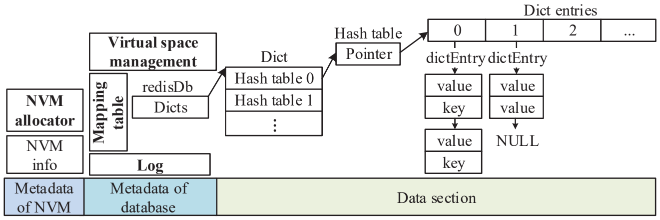


Fig. 3. The data structures of Redis in memory.

update operation of conventional in-memory KVDBs may write the data and command four times and cause large overhead.

The challenges and opportunities for ensuring consistency and durability of UDORN are different from these of existing in-memory databases on block devices. On one hand, the hardware techniques of processors may damage the reliability of data on NVM. For example, out-of-order execution may reorder the execution of instructions in an operation and caching may delay the writes to the destined physical NVM cells. Suppose a system crash or power failure occurs in an operation, the operation can be either incorrect or unsaved on NVM, which leads the database into an inconsistent state. Therefore, the operations in the framework ensure data consistency and durability on NVM by explicitly flushing related cachelines and memory barriers. It can be done by calling CPU instructions, such as Intel's *clflush*, *clflushopt* (a more efficient version of *clflush*), *mfence*, *clwb*, and *PCOMMIT* (to force stores out to NVMM) [32].

On the other hand, the operations can take advantages of the atomic operations to simplify the consistency mechanism. For example, the data on NVM that is less than 64 bytes can be updated without disturbing other data via atomic in-place update operations [33], such as the *cmpxchg8b/16b* instructions of Intel processors [34]. Taking advantages of these atomic operations, a unified database on raw NVM can update a record by updating data with copy-on-write (COW) and modifying related pointer with an atomic operation.

With these features in mind, we design a COW-based mechanism for the operations in the proposed design of UDORN. An operation directly writes the affected data on NVM once and updates the affected metadata via an atomic operation. We will illustrate the mechanism via a case study in Section IV-C.

IV. A Case Study of UDORN Based on Redis

In this section, we show how to apply the proposed design of UDORN to existing in-memory key-value database (KVDBs) via a case study. Concretely, we implement a persistent in-memory KVDB, namely UDORN, based on the open-source Redis [9].

A. Management of NVM

The proposed UDORN manages the physical space of NVM device by itself. The physical NVM is managed in kernel space. To reuse the paging mechanisms in existing computer systems, we still divide the NVM physical space to pages. The size of a page is set to 4KB.

The layout of NVM with UDORN is shown in Fig. 3. The metadata of NVM is stored in the headmost of NVM. The metadata of NVM includes a NVM allocator and the basic information of NVM (NVM info), such as the size of NVM.

Since the key-value pairs may vary in sizes arbitrarily, the allocator of UDORN provides space management not only for full page requests but also smaller granularity memory request. All the free pages are managed by free lists. UDORN allocates free pages from free lists when the requested space is more than one page. For smaller granularity space allocation requests, UDORN calls a fine-grained allocator that is similar to the slab allocator of Linux kernel.

B. Persistent Metadata Structures of Database

There are three main metadata structures used to locate the database in launching UDORN. The first metadata structure is a mapping table. The mapping table organizes the physical space engaged by the datasets and organization structures of UDORN, such as dictionaries and hash tables shown in Fig. 3. When UDORN is launched, the mapping table is embedded to the corresponding process page table of UDORN. Then the process of UDORN is capable of accessing the database in user space.

The second metadata structure is the virtual space management of UDORN. We reserve a virtual address space with the size equal to the maximum size of the database, e.g., the size of NVM. The virtual space management uses the same approach of the buddy system in Linux. Each time the database submits an update request, UDORN allocates a segment of free virtual space for the request. The size of the free virtual space is equal to or aligned to the last page of the requested physical memory. When a key-value pair is deleted, the corresponding free space will be reclaimed.

Finally, a metadata structure is needed to indicate the addresses of organization structures even though they are exposed to the address space. For UDORN, the structure *redisDB* stores the dictionaries that indexes all the data, as shown in Fig. 3,. Thus, we maintain *redisDB* in the metadata of UDORN.

C. Consistency of Operations

Moreover, we need a mechanism to ensure the consistency of updates since they are directly performed on the persistent database. In UDORN, there are two typical schemes to make the updates of in-memory database consistent on the backup image located on secondary storage: Append Only File (AOF) and Snapshot (namely, RDB). The AOF scheme logs every operation that modifies the dataset in memory. Then, the operation is performed on the database. AOF protects every update operation but causes large I/O costs. The RDB scheme produces snapshots of the in-memory dataset when a condition is satisfied, such as a period of time. RDB has less I/O costs but cannot ensure the consistency of updates if a failure occurs before the backup condition is met. Both AOF scheme and RDB scheme write new value twice: once on the storage for backup and once in-memory for processing.

In UDORN, we propose a journaling scheme to backup the update operations of dataset. Different from existing approaches of Redis, UDORN only writes new value once taking advantages of the address space of NVM.

Algorithm 1 Implementation of SET operation.

Allocate free virtual space for the key-value pair;
Allocate a log entry for the operation and write ξ;
`mfence()`; `clflush(`ξ`)`; `mfence()`;
Allocate free NVM space and setup the corresponding mapping table;
Write new $(key, value)$ to newly allocated space;
Modify the dictEntry ε with the pointer to the new $(key, value)$;
`mfence()`; `clflush(`ε`)`; `mfence()`;

UDORN initiates a log in NVM. Each log entry can store the operation type, address, and the size of a key-value pair, the affected key. UDORN performs *SET* following the procedures shown in Algorithm 1. Firstly, UDORN records the address space, the size of the key-value pair, and type of operation in a new log entry ξ. The log entry is made durable on NVM via *clflush*() and *mfence*(). Then, the new key-value pair $(key, value)$ is written once to a newly allocated space. The corresponding dictEntry is also updated. Suppose a crash occurs in step 4-6, the NVM space of $(key, value)$ can be reclaimed during recovery according to the address and the size recorded in ξ. The dictEntry ε can be reset via the key stored in ξ. Finally, the updates are made durable.

Algorithm 2 Implementation of DELETE operation.

```
Allocate a free log entry (ξ) and write ξ;
mfence(); clflush(ξ); mfence();
Free the affected key-value pair;
Clear the related dictEntry ε;
mfence(); clflush(ε); mfence();
```

For *Delete* operation shown in Algorithm 2, UDORN simply records the operation type and the affected key in log entry durably before the key-value pair is removed. During recovery, UDORN redo an unfinished *Delete* operation if the log entry has been made durable.

V. Performance Evaluation

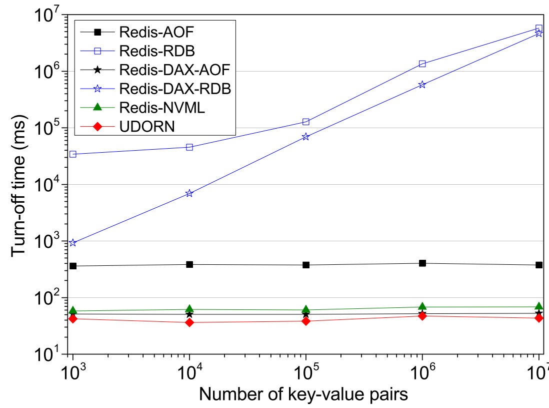
In this section, we present experimental results of UDORN against the original Redis with various configurations. For the impact of file system over the performance of Redis, we deploy Redis in both traditional scenario (i.e., a HDD managed with Ext4) and an in-memory scenario, such as a ramdisk (DRAM) managed by Ext4 in DAX [16] mode (Ext4-DAX). The two persistence modes of Redis, i.e., AOF and RDB, are measured in the experiments. Furthermore, we also compare UDORN with the exended Redis using the NVM Library [18], denoted by "Redis-NVML".

The experiments are conducted on a workstation equipped with 128GB DRAM and a 2.6GHz Intel® Xeon® E5-2650 eight-core processor. For in-memory file system Ext4-DAX and UDORN, we configure 64GB memory as data storage. The rest 64GB memory is used as the main memory of the system.

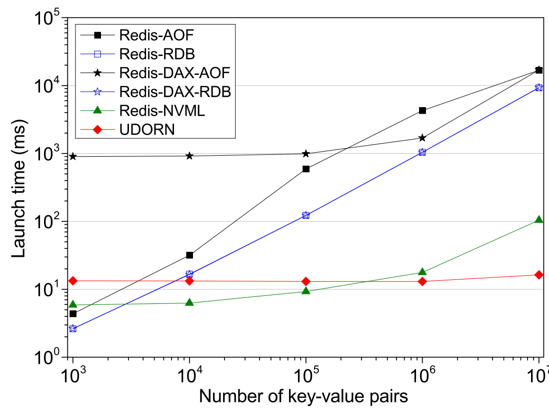
We begin by evaluating the efficiency of launching and turning off the database. The experimental results are shown in Fig. 4. Fig. 4(a) shows that the turn-off time of UDORN reaches 100 thousand times faster than that of Redis in RDB mode, either on HDD (Redis-RDB) or memory (Redis-DAX-RDB). Due to the backup operations, the time for shutting down Redis in RDB mode is increased along with the size of database. The turn-off times of UDORN and Redis in AOF mode stay steady on all scales of the database. However, UDORN still reaches 8 times, 24.8% faster than Redis-AOF and Redis-DAX-AOF, respectively. It is because that UDORN avoids the cost for traversing I/O routines and recycling the in-memory database. Furthermore, because of Redis-NVML saves states in the memory-mapped files of NVM by the NVM Library, UDORN is 53.7% faster than Redis-NVML on average.

Fig. 4(b) shows that the launch time of UDORN reaches 1000 times and 6 times faster than that of original Redis and Redis-NVML in the best case. The more key-value pairs are stored in a database of the original Redis or Redis-NVML, the more time it takes to startup the database. On the contrary, the launch time of UDORN stays steady on all scales of the database. It is because that UDORN simply exposes the persistent database to process address space via a mapping table rather than constructing another database in main memory.

Next, we evaluate the performance of database workloads via *redis-benchmark* [19]. For each workload, *redis-benchmark* simulates 10 parallel clients to submit 100000 requests. The workloads can be divided into two categories: the ones involving update operations and the ones that only involve query. Concretely, the former category workloads include SET, INCR, LPUSH, LPOP, MSET, and SADD. The other category workloads include SPOP, GET, PING_INLINE, PING_BULK, and LRANGE 100 to 600. The experimental results are shown in Fig. 5 and Fig. 6.



(a) Turn-off time.



(b) Launch time.

Fig. 4. Comparison of turn-off time and launch time.

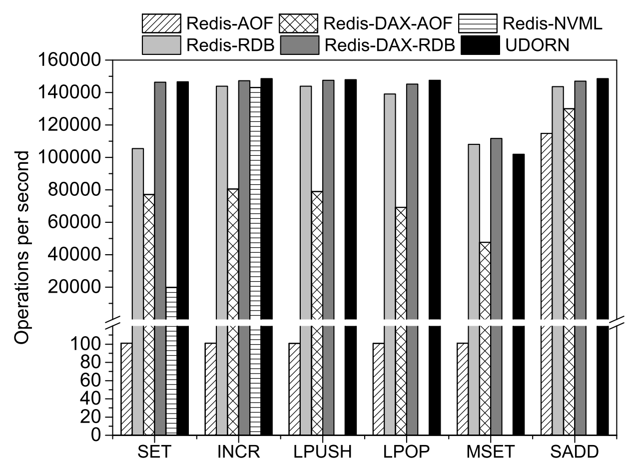


Fig. 5. Experimental results of workloads with updates.

Fig. 5 shows that the operations per second (OPS) of UDORN significantly surpasses these of Redis-AOF and Redis-DAX-AOF for the workloads involving update operations. Compared with Redis-AOF, the OPS of UDORN is 1449, 1471, 1465, and 1461 times higher for workloads SET, INCR, LPUSH, and LPOP, respectively. Compared with Redis-DAX-AOF, the OPS of UDORN is 90%, 84.5%, 87.2% and 112.9% higher for workload SET, INCR, LPUSH, and LPOP, respectively. Meanwhile, UDORN provides same consistency with Redis-AOF, i.e., they all can recovery to the latest update operation. It is because that UDORN writes new data only once and avoids the cost for backing up operations through I/O routines.

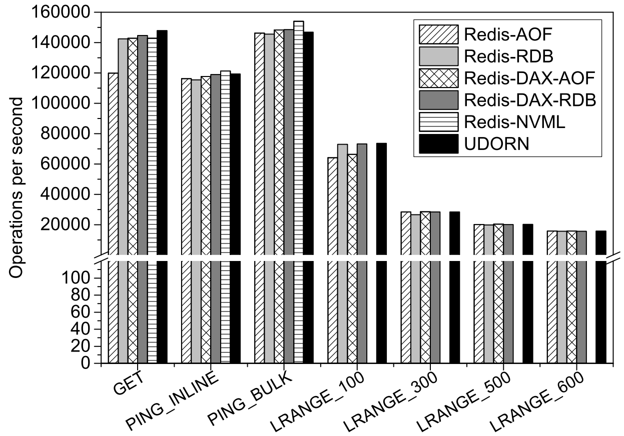


Fig. 6. Experimental results of workloads without updates.

When Redis uses the RDB mechanisms, the performance of UDORN is almost the same as Redis on HDD (Redis-RDB) or memory (Redis-DAX-RDB). It is because that the updates operations of Redis in RDB mode are performed temporarily in main memory without backing up on storage yet. Thus, they cannot provide the same consistency insurance with UDORN. For the workloads that only query the database, the OPSs of UDORN are almost the same (the variation is less than 5%) as these of the four configurations of Redis, as shown in Fig. 6. It means that UDORN has no additional overhead for general query operations.

Because of Redis-NVML only support string values and cannot delete key-value pairs, it can only run five workloads, including SET, INCR, GET, PING_INLINE, and PING_BULK. The OPS of UDORN is 6.4 times higher than Redis-NVML for SET. It is because that Redis-NVML logs the operations by the log file of *libpmemlog*. The implementation of INCR operation in Redis-NVML uses DRAM space and has no persistence guarantees, so it shows same performance as Redis-RDB. For the rest of three workloads, Redis-NVML and UDORN also have almost the same OPS (less than 5% variation) for they perform the workloads in similar way.

VI. Conclusion

This paper presented a new design framework of persistent in-memory KVDB, UDORN, when NVM is deployed as storage. UDORN uses a persistent database to achieve the functions of both the in-memory database and persistent image of conventional in-memory KVDBs. Specifically, users can directly access the persistent database on NVM via process virtual address space. The persistent database employs logs and dedicated memory management of NVM to ensure consistency and persistency. A persistent KVDB is implemented based on the framework of UDORN. Experimental results show that UDORN can achieve more than 6 times, 1400 times, and 84% performance improvement over Redis using the NVM Library, Redis on HDD, and Redis using in-memory file system, respectively.

References

[1] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, “Architecting to achieve a billion requests per second throughput on a single key-value store server platform,” in *ISCA*, 2016, pp. 476–488.

[2] N. Agrawal, A. Aranya, and C. Ungureanu, “Mobile data sync in a blink,” in *USENIX HotStorage*, 2013, pp. 3–3.

[3] W. H. Kim, B. Nam, D. Park, and Y. Won, “Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split,” in *USENIX FAST*, 2014, pp. 273–285.

[4] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, “Graph/z: A key-value store based scalable graph processing system,” in *IEEE CLUSTER*, 2015, pp. 516–517.

[5] https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, 2015.

[6] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” in *ISCA*, 2009, pp. 14–23.

[7] E. Kltrsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, “Evaluating stt-ram as an energy-efficient main memory alternative,” in *IEEE ISPASS*, 2013, pp. 256–267.

[8] S. S. Parkin, M. Hayashi, and L. Thomas, “Magnetic domain-wall racetrack memory,” *Science*, vol. 320, no. 5873, pp. 190–194, Apr. 2008.

[9] Redis, https://redis.io/.

[10] Aerospike, http://www.aerospike.com/.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better i/o through byte-addressable, persistent memory,” in *ACM SOSP*, 2009, pp. 133–146.

[12] X. Wu, S. Qiu, and A. L. Narasimha Reddy, “Scmfs: A file system for storage class memory and its extensions,” *ACM Transactions on Storage*, vol. 9, no. 3, pp. 1 – 11, 2013.

[13] J. Xu and S. Swanson, “Nova: A log-structured file system for hybrid volatile/non-volatile main memories,” in *USENIX FAST*, 2016, pp. 323–338.

[14] E. H. M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, “A new design of in-memory file system based on file virtual address framework,” *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 2959–2972, Oct 2016.

[15] J. Ou, J. Shu, and Y. Lu, “A high performance file system for non-volatile main memory,” in *EuroSys*, 2016, pp. 12:1–12:16.

[16] M. Wilcox, “Add support for nv-dimms to ext4.” https://lwn.net/Articles/613384/.

[17] http://pmem.io/nvml/.

[18] https://libraries.io/github/pmem/redis.

[19] *redis-benchmark*, https://redis.io/topics/benchmarks, 2016.

[20] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in *ACM SOSP*, 2007, pp. 205–220.

[21] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “Silt:a memory-efficient, high-performance key-value store,” in *ACM SOSP*, 2011, pp. 1–13.

[22] B. Debnath, S. Sengupta, and J. Li, “Flashstore: high throughput persistent key-value store,” *Proceedings of the Vldb Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[23] G. Lu, Y. J. Nam, and D. H. C. Du, “Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash,” in *MSST*, 2012, pp. 1–11.

[24] J. S. Ahn, C. Seo, R. Mayuram, and R. Yaseen, “Forestdb: A fast key-value storage system for variable-length string keys,” *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 902–915, 2016.

[25] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, “Nvmkv: a scalable, lightweight, ftl-aware key-value store,” in *USENIX ATC*, 2015, pp. 207–219.

[26] L. Lu, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Wisckey: Separating keys from values in ssd-conscious storage,” *ACM Transactions on Storage*, vol. 13, no. 1, p. 5, 2017.

[27] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, “Exploring storage class memory with key value stores,” in *The Workshop on Interactions of Nvm/flash with Operating Systems and Workloads*, 2013, p. 4.

[28] L. Marmol, J. Guerra, and M. K. Aguilera, “Non-volatile memory through customized key-value stores,” in *USENIX HotStorage*, 2016.

[29] O. database online documentation 11g release 1, http://docs.oracle.com /cd/B28359_01/install.111/b32002/install_overview.htm#LADBI187.

[30] A. Thomson and D. J. Abadi, “The case for determinism in database systems,” *VLDB*, vol. 3, no. 1-2, pp. 70–80, Sep. 2010.

[31] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, “Rethinking main memory oltp recovery,” in *IEEE ICDE*, March 2014, pp. 604–615.

[32] “Intel architecture instruction set extensions programming reference,” https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2016.

[33] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in *EuroSys*, 2014, pp. 15:1–15:15.

[34] “Intel64 and ia-32 architectures software developer’s manual. (vol 2, chp 3.2),” 2014.