Efficient Reboot-Based Recovery of In-Memory Databases

Yuto JUMONJI¹, Nonmember and Hiroshi YAMADA¹(¹), Member

SUMMARY Reboot-based recovery is a simple but powerful method to recover applications from failures and unstable states. Reboot-based recovery faces a challenge to apply it to a new type of applications, in-memory databases (DBs). Unlike legacy applications, since reboots in-memory DBs lose memory objects including key-value pairs and DB blocks, it is required to restore them, causing severe performance degradation after the reboot. This paper presents an approach that allows us to perform reboot-based recovery of in-memory DBs with lower performance degradation. Our key insight is to decouple data content objects from all the memory objects. Our approach treats data items as data content objects, preserves data content objects on memory across reboots, and enforces restarted in-memory DBs to attach them. To show the effectiveness of our approach, we elaborate the idea into two real-world DBs, MyRocks and memcached. The prototypes successfully mitigate performance degradation after their reboot-based recovery.

key words: reboot-based recovery, software rejuvenation, in-memory DBs

1. Introduction

Reboot-based recovery is a simple but powerful method to recover applications from failures and unstable states. The reboot-based recovery is to restart the target applications to refresh all the internal states on memory. Periodically performing application restarts, a.k.a. software rejuvenation [19], [24], reclaims stale or leaked resources such as unreleased memory objects and unclosed descriptors, resulting in preventing application crashes proactively. Reboot-based recovery is also used to recover applications from crashes and hangs. We can conduct reboot-based recovery without analyzing the root causes and thus it often becomes the only remedy against error states and failures for end users. To extend the applicability of reboot-based recovery due to its usefulness, researchers have explored effective software mechanisms for various software layers such as java applications [13], [36], operating systems [12], [17], [27], [35], [38], and hypervisors [28]–[30].

Reboot-based recovery faces a challenge to apply it to a new type of applications, in-memory databases (DBs). In-memory DBs, such as MyRocks [32] and memcached [3], manage a considerable number of data items using several tens to hundreds of GB memory to reduce accesses to storages. The reboot-based recovery is effective for real-world in-memory DBs because they suffer from error states and failures stemming from software bugs [18]. However, restarting in-memory DBs causes a non-trivial cost compared to legacy stateless applications such as web servers. In-memory DBs typically allocate a large amount of memory and manage numerous data items such as key-value (KV) pairs and DB blocks on their memory space. Since restarting in-memory DBs loses all the memory objects including data items, the performance of restarted in-memory DBs is degraded severely. For example, rebooting memcached, which manages KV pairs in its address space, loses all the KV pairs and thus involves their restoration from data sources, such as storages and its replica, whose accesses are much slower than memory accesses. In the market places, in-memory DB restarts are known to be costly. A research literature reports that restarting only 2% of the servers at a time in Facebook prolongs the restart duration to about 12 hours, during which users see only partial query results [22].

This paper presents an approach that allows us to perform reboot-based recovery of in-memory DBs with low performance degradation. Our key insight is to decouple data content objects from all the memory objects. We specifically preserve data content objects on memory while refreshing the other objects, referred to as management objects that are a major source of error states [21]. Our approach treats data items such as DB blocks and KV pairs as data content objects and keeps them on memory across in-memory DB’s restarts. To safely attach data content objects after reboot-based recovery, the approach also protects them against incorrect modification caused by the error propagation of the fault. The main technical challenge is how we apply this idea to real-world in-memory DBs that employ sophisticated memory manager such as slab allocator. In this paper, we elaborate the idea into two real-world DBs, MyRocks and memcached.

This paper substantially extends our previous work [25]. The main differences are specifically 1.maturing the concept and description of our approach, 2.applying our idea to new applications: MyRocks and multi-threaded memcached, 3.conducting more experiments to show the effectiveness of our approach, and 4.surveying more existing work to clearly show the contributions of our work. This paper makes the following contributions:

• This paper presents a reboot-based recovery method tailored to in-memory DBs. Our approach is characterized as follows. First, our approach mitigates performance degradation after reboot-based recovery of in-

---

¹The authors are with the Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan. 
a) E-mail: hirosiy@cc.tuat.ac.jp
DOI: 10.1587/transinf.2020ZDP7501
memory DBs. Second, the approach enforces restarted in-memory DBs to safely attach in-memory data items by protecting them against error propagation of the faults. Lastly, the approach causes less runtime performance and space overhead. We note that the restart time of our approach, which depends on the number of data content objects, is much shorter than the conventional warm-up time that involves heavy accesses to external resources such as networked replica and storages (Sect. 3).

- We conduct two case studies using real-world in-memory DBs: MyRocks and memcached. These in-memory DBs have their memory managers such as slab allocator and LSM-tree based block manager. The case studies show that our software mechanisms can be successfully integrated into both memory managers and control data content objects for their reuse with less perturbation of runtime performance (Sect. 4 and 5).

- We implemented prototypes by extending MyRocks and memcached, and carried out experiments to demonstrate their effectiveness. The experimental results reveal that the prototypes shorten more than 90% memory warm-up time compared to the regular reboot, and their runtime overhead is less than 9%. Also, the prototypes detect failures and successfully recover from them under our fault injections (Sect. 4 and 5).

2. Background

2.1 Reboot-Based Recovery of In-Memory DBs

Reboot-based recovery is effective for in-memory DBs because they suffer from software bugs leading to crashes, hangs, and error states. For example, a community reports that memcached causes memory leaks and resource leaks of descriptors that are recoverable by reboot-based recovery [2], [4]. A reported issue is that memcached never releases a temporal memory buffer, leading to a memory leak. Memcached allocates a memory buffer, dump_buf, with malloc() but the buffer is not freed in an error-handling path. Reboot-based recovery releases these buffers without fixing the code.

The advanced memory management of memcached causes a negative effect. Slab calcification, a widely-known negative phenomenon, makes the slab allocators fail to allocate the appropriate size of slabs under workload changing [23]. For example, there are two classes of slabs to receive a sequence of requests. In the example, the sequence of items for writing into Class 1 is “abcabcabc...”, and the sequence into Class 2 is “123456789...”. We also assume that each slab holds only one item in both classes for the sake of simplicity, and there are four slabs. The workload consists of the combined access pattern, “1abc2abc3abc4abc5a...”. If Class 1 and 2 can accommodate 3 and 1 items respectively, the total cache hit ratio becomes 10. In this case, Class 1 caches the whole working set (a, b, c) after the cold start period. However, the situation is different if the slab number of Class 1 and 2 is fixed before this access pattern comes. If Class 1 and 2 can accommodate 2 and 2 items respectively due to the slab allocation defined by the older workload, the total cache hit ratio is 0. Class 1 cannot cache the whole working set and unfortunately evicts the next accessed item. Even if Class 2’s hit ratio is 0, we cannot move slab slots across the slab class. We can say that slab calcification happens. Reboot-based recovery eliminates slab calcification since the reboots clear all the slabs and thus the slab allocation becomes suitable for the current workload.

Another example is that MyRocks is also reported to cause memory leaks and resource leaks [7], [8]. A bug report mentions that a memory leak is caused by unnecessary memory allocation for a transaction rollback. Specifically, MyRocks unnecessarily prepares a malloc()-based memory buffer, Rdb_transaction in a code path. This memory buffer is never freed and thus leads to a memory leak. Also, MyRocks inherits MySQL bugs since its backend is MySQL that consists of a large body of source code. Restarting MyRocks is an effective way to eliminate such a memory leak since all the memory objects are cleaned and reconstructed.

2.2 Problems

In this paper, we try to answer the following question: How can we efficiently conduct reboot-based recovery of in-memory DBs? Reboot-based recovery implicitly assumes that target applications are so stateless that their memory objects can be quickly restored after reboots. Since in-memory DBs manage a tremendous number of memory objects such as KV pairs and DB blocks on their memory space and rebooting them loses all the memory objects, the running states’ restoration is costly, causing significant performance degradation. The performance of the restarted in-memory DBs becomes much lower than their peak performance until the restoration of the memory objects from back-end storages and/or their replicas, which typically requires non-trivial time, completes. A research literature reports that restarting only 2% of the servers at a time in Facebook prolongs the restart duration to about 12 hours, during which users see only partial query results [22].

Approaches to restoring memory contents of in-memory DBs such as using a memory snapshot in disks [31] or keeping data item objects across the reboot-based recovery [5] are useful to mitigate the performance degradation but pose another challenge: error propagation of the fault can damage memory objects, failing to restart in-memory DBs correctly. Since in-memory DBs typically utilize tens or hundreds GBs of memory, error propagation caused by software bugs is easy to modify the memory objects relatively to the legacy stateless applications [33]. When memory objects for data items are damaged and then stored to back-end storages, the restarted in-memory DBs restore the damaged objects after reboot-based recovery. For example, MyRocks returns unexpected values to clients if the memory objects loaded from an SSTable are modified illegally and written back to it.
3. Proposal

This paper presents an approach to allow us to perform an efficient reboot-based recovery of in-memory DBs. The approach has the following design goals:

- **Mitigate performance degradation after reboot-based recovery:** The regular reboot-based recovery removes all of the memory contents of in-memory DBs, causing performance degradation until the data items have been restored. Our approach quickly restores data items to shorten the performance degradation phase.

- **Minimize resource overhead at runtime:** We design our reboot-based recovery to perform on a single machine. Replication of the target in-memory DB is a useful technique to avoid performance degradation during restarting an in-memory DB instance. This, however, involves an additional administration cost for replicas and is sometimes impossible to use in resource-restricted environments such as nodes in the edge computing.

- **Avoid long restart time:** Our approach offers short downtime for restarting the target in-memory DB, similarly to regular restarts. We do not employ a snapshot-based approach that dumps memory contents for restoration. Since the memory footprint of in-memory DBs is typically tens to hundreds of GB, it is quite a time-consuming task to take and restore from a snapshot.

The key behind our approach is to decouple data content objects from all the memory objects. We preserve data content objects on memory while refreshing the other objects, referred to as management objects that constitute a significant error state source. Our approach treats the KV pairs and DB blocks as data content objects and preserves them on memory across reboots. For example, in memcached, we can treat slabs where KV pairs are stored as data content objects and the other objects as management objects. The rebooted memcached regains the peak performance for a short time by attaching the preserved in-memory slabs.

An overview of our approach is shown in Fig. 1. The regular reboot-based recovery clears all the memory contents of the target in-memory DBs. The performance degradation continues until all the data items have been restored from other resources such as storages and other in-memory DBs. This interval becomes longer as the total memory size of data items is bigger. On the other hand, our approach enforces the in-memory DBs to attach data content objects after their restart while the other memory objects are recreated. Since the in-memory DBs can restore the data items from memory on the same machine, the performance degradation is much shorter than the regular one. Also, our recovery refreshes the other memory objects such as leaked descriptor tables and unreleased buffers, a.k.a. management objects.

Fig. 1 Reboot-based recovery overview

Also, we protect data content objects against error propagation of the faults such as software bugs. Our approach switches read and write permission of data content objects in a fine-grained manner, borrowing the idea of kernel-level file cache protection [15]. Almost all data content objects are read-only in our approach, while only the data content objects to be updated are switched to the write-mode. For example, when in-memory DBs receive an update request, our mechanism changes the read-only corresponding data items to writable. And then, it resets the items to read-only just after their updates. By doing so, we try to lower the possibility of illegal modification of data items as much as possible.

The recoverable software errors and failures of our approach almost inherit ones of the regular reboot-based recovery. As well as the application reboots, our approach can refresh stale management objects and be used for crashes and hangs. The approach cannot reclaim error states propagated in data content objects since the restarted in-memory DB reuses the data content objects after reboots. The errors occur hardly; data content objects are read-only basically and code paths for updating them are short, which is the same design philosophy as OS buffer cache protection [15]. It can also not cover the unrecoverable software errors and failures of the regular reboot-based recovery such as deterministic software/hardware failures.

The idea behind our approach is simple but poses a practical and technical challenge: *How can we apply our approach to real-world in-memory DBs?* The internal data structures of the modern in-memory DBs are complicated for multi-threading, sophisticated memory manager, and various data type supports. To show our approach is applicable to real-world in-memory DBs, we carry out two case studies with open-sourced in-memory DBs: *MyRocks* and *memcached*. In Sect. 4 and 5, we describe design and implementation of our approach on these in-memory DBs in details.
4. Case Study: MyRocks

4.1 Memory Management

MyRocks [1] is a MySQL [6] instance whose back-end engine is RocksDB [10]. RocksDB employs a log-structured merge-tree (LSM-tree) [34] data structure for its KV pairs management. The LSM-tree offers indexed accesses to KV pairs using multi-level tables. New KV pairs are first stored in the level-0 table. When the level-0 table is filled up with KV pairs, all of the values are sorted by keys and are written to a level-1 table resident on storages such as HDD and SSD. When the level-1 tables are filled with key-value pairs, the contained values are sorted and merged into a level-2 table. This is continuously performed in a multi-level manner.

The essential data structures in MyRocks are SSTable, MemTable, and Block Cache [9], shown in Fig. 2. The SSTable is a data structure used as level-1 to N tables for storing KV pairs on storages such as HDD and SSD. MyRocks inserts multiple KV pairs into an SSTable and writes back the compressed SSTable into storages. When a value on the storage is requested, MyRocks fetches the corresponding SSTable and returns it after decompressing the SSTable. MemTable and Block Cache are in-memory objects for performance improvement. MemTable is the level-0 table and consists of recently stored KV pairs. In MemTable, the KV pairs are managed by a skiplist for quick access. When the number of the stored data items is over the threshold, the current contents of MemTable is written back into the storage as an SSTable. Block Cache is an in-memory cache of KV pairs for reducing the latency of read requests. MyRocks keeps the configured number of the decompressed SSTable in Block Cache.

4.2 Preserving Data Content Objects

In this work, we treat Block Cache as data content objects and the other objects as management objects, shown in Fig. 3. After the reboot, the performance of MyRocks is degraded until it restores the Block Cache contents by reading the target SSTables from storages. Our mechanism keeps the Block Cache contents on memory across reboots and enforces the restarted MyRocks to attach them. The other objects such as MemTable and network states are recreated, which is the same as the regular reboot.

To preserve Block Cache across MyRocks’ restart, our mechanism, running inside the MyRocks memory manager, leverages a shared memory feature supported by commodity OSes. The shared memory feature, originally designed for inter-process communication, allows processes to map a piece of memory into their own address spaces. The underlying OS manages the shared memory regions and processes can control them via system calls. The shared memory region is kept even after a process that creates it finishes. Processes can map a shared memory region formed by an already dead process into their address spaces.

Our mechanism places Block Cache on shared memory. The mechanism allocates shared memory when an SSTable block is fetched from storages and decompressed. It puts the decompressed block contents on a shared memory with its metadata that consists of the cache key length, block length, and the cache key. It saves the metadata and blocks on shared memory serially. The block cache metadata, managed on regular memory by MyRocks, points to blocks on shared memory. After MyRocks has restarted, our mechanism recreates the block cache by fetching SSTable blocks from the shared memory. The mechanism specifically reads our metadata and reconstructs the Block Cache metadata based on it.

We carefully design our mechanism to avoid interfering with existing recovery functionalities such as transaction mechanisms. Since our software mechanism only changes the controls of Block Cache objects and does not modify their metadata management, the functionalities of MyRocks works as-is. For example, the existing mechanism deals with transaction failures, even if our mechanism is running.

4.3 Memory Protection

To protect Block Cache against the error propagation of the faults, our mechanism sets the shared memory region’s permission to read-only as long as possible while turning into writable only at Block Cache updates. To do so, our prototype invokes mprotect() system call when initializing and updating a block in Block Cache, called cache block. Specifically, the prototype changes the target cache block’s memory page to read-only after the initialization of a new cache block. When updating a cache block, it changes the page permission to writable, updates the block, and then re-
sets the permission into read-only again.
In addition, we calculate and save the checksum of each cache block to reuse the Block Cache more robustly. When a cache block is initialized or updated, the prototype calculates its checksum and preserves it on a shared memory region. It checks the checksum every the first access to a restored cache block after reboots.

Note that our mechanism does not always protect Block Cache against overwrites stemming from the faults’ error propagation. The error propagation can damage cache blocks when their permission is switched to be writable. Our protection mechanism tries to lessen the possibility of error propagation damages by shortening the interval during which a cache block is writable as much as possible. We also validate Block Cache contents by using the checksum. We believe that our mechanism safely reuses the Block Cache. Our experiments do not observe that fault injections cause our prototype to use damaged cache blocks by evading its protection.

4.4 Experiments

We prototyped the approach on MyRocks 5.6 based on RocksDB 5.18.0 and conducted experiments to show its effectiveness. We run the prototype on Xeon E7-4870 2.40 GHz with 1 TB of memory. We also run Ubuntu 18.04 LTS on the machine. In the experiments, we answer the following fundamental questions: 1) how is the runtime overhead of our approach?, 2) how does our approach mitigate performance degradation after reboot-based recovery?, and 3) how can our approach recover from failures?

4.4.1 Runtime Overhead

To show how runtime overhead our prototype incurs, we measure the throughput of both vanilla and our MyRocks under YCSB [16] benchmark. We ran all the default workload of YCSB, a, b, c, d, e, and f. The workload types are update heavy(a), read mostly(b), read-only(c), read latest(d), short ranges(e), and read-modify-write(f).

The result is shown in Fig. 4. We can see that the runtime overhead of the prototype is up to 8%. The overhead is negligible in read-intensive workloads since our prototype does not interfere with the read operations. On the other hand, the overhead is more in write-intensive workloads. This comes from our memory protection mechanism that switches Block Cache pages’ permission to every cache block update. The metadata for each block on MyRocks consists of ID (20 bytes) and length (4 bytes). The size of metadata is much less than the block (Decompressed SSTable) size that is 8 K bytes.

4.4.2 Throuput across Reboot-Based Recovery

To demonstrate our prototype mitigates performance degradation, we measure the throughput of the prototype after MyRocks restarts. We prepare an intentional workload that stores 20 items, each of which is 1024 bytes, and then read them randomly. We perform our rejuvenation after the read workload runs for 9 hours. For comparison, we also perform the regular restart of MyRocks under the same situation.

Figure 5 shows the throughput of MyRocks. From the figure, we can see that the prototype successfully mitigates performance degradation after the reboot. The regular reboot takes 8.5 hours to perform full throughput (3900 queries per second) since MyRocks reconstructs the Block Cache from scratch, which means that a large number of storage accesses occur. On the other hand, the prototype performs its full throughput in 54 seconds after the reboot.

4.4.3 Fault Injection

To demonstrate our protection mechanism protects data items against illegal writes, we perform software fault injection into the running prototype. We perform 100 updates to a random page of the shared memory region of our prototype. We also analyze the prototype behavior when the faults manifest.

The result is that the prototype terminates in a fail-stop manner or keeps running under our fault injection. The fail-stops of the prototype occur in 98 faults due to segmentation faults. All the segmentation faults come from write access to read-only Block Cache. The other cases are that the prototype continues to run since the injected updates modify unused memory objects.

5. Case Study: Memcached

5.1 Memory Management

Memcached is a multi-threaded in-memory KV store and its index is based on a hash table. The multiple threads handle requests such as set and get concurrently. When we store a
KV pair via a set command, memcached allocates a memory region to save the pair, calculates a hash value of the key, and registers it to the hash table. Memcached searches the target value from the hash table in a get command.

In memcached, the memory management is performed by its slab allocator, shown in Fig. 6. Instead of allocating memory on an item basis, the slab allocates slabs, each of which is 1 MB of memory by default, and divides them into chunks. Each slab consists of a fixed size of chunks and the chunk size is different among slabs. To avoid memory fragmentation as much as possible, the slab allocator preserves a value into a chunk with the same size as or closest to the value size. When a value is deleted, memcached returns its chunks to the slab allocator instead of freeing the value's memory region.

5.2 Preserving Data Content Objects

We treat slabs as data content objects and the other objects as management objects. Memcached stores values in slabs and its reboot clears all the slabs, causing performance degradation until they are restored. Our mechanism keeps all of the slabs on memory across reboots and enforces the restarted memcached to attach them. Also, it recreates the other objects such as hash chains and descriptor tables, which are a significant part of error sources.

Similar to the previous case study, our mechanism places slabs and our original metadata on shared memory regions, shown in Fig. 7. Instead of slabclass, which is used as slab’s metadata in memcached, we use the original metadata that consists of the slab’s item size and number to avoid frequent permission changes as described in the next section. The original metadata is preserved to divide the slab regions into items correctly.

In memcached’s initialization phase, our mechanism, running in the slab allocator, allocates shared memory. When memcached receives set requests, the mechanism prepares a slab and our original metadata of the KV pair on the shared memory. After that, memcached stores it on the slab. When memcached has been restarted, our mechanism restores slabs by tracing our metadata and chaining slabs on the shared memory. And then, the mechanism reconstructs other KV related data structures such as hash tables.

We note that our approach can mitigate slab calcification [14], [23]. Instead of restoring all the slabs, the mechanism randomly releases some of the slabs in each slabclass. By doing so, more new KV pairs can be set into the appropriate slab class. The best number of released slabs depends on incoming workloads and performance guarantees, and exploring this is out of the scope of this paper.

5.3 Memory Protection

Similar to the MyRocks case study, our mechanism changes the permission of shared memory pages in a fine-grained manner to protect slabs against the error propagation of faults. The mechanism keeps slab and our metadata read-only while their permission is switched to writable on their updates by issuing mprotect() system call. Specifically, the prototype sets a page containing a slab and its metadata to read-only after their initialization. It puts slabclass objects on regular memory due to their frequent updates, reducing permission changes. When setting or deleting a data item, it changes the target slab region to writable, updates it and its slabclass, and resets the page permission to read-only. Since the size of each slab is 1 MB which is the default page size $\times 250$, this is suitable for the mprotect-based protection that is performed at the page size granularity.

Our mechanism calculates each KV’s checksum and stores its multiple copies on the different regions of shared memory to enhance the slab reuse’s robustness. Figure 7 overviews the slab restoration of our mechanism. Our mechanism first checks the stored checksum values when reusing a preserved value and attaches the KV pairs when their checksum is validated. In the current version, the mechanism judges the restored value is valid if only one checksum is equal to the corresponding item’s hash value.

We make use of the checksums to optimize the KV delete operation. When a delete operation is received, our mechanism never modifies the corresponding slab; it only updates the hash table and slabclass for KV’s metadata, and invalidates its checksums. By doing so, we can skip the permission change of slab. In the restoration, our mechanism also salvages KV pairs from slabs whose checksum is valid. In doing so, the mechanism keeps the shared memory pages containing checksums writable. We believe that this does not affect the robustness of the slab restoration very much because all of the checksum copies are rarely damaged at once and the mechanism can use the correct checksum using the undamaged copies. In the current prototype, we prepare 3-copies of the checksums to mitigate checksum pollution by error propagation.
5.4 Experiments

We prototyped our mechanisms on Memcached 1.4.25 and conducted experiments to show its effectiveness. We ran the prototype on Ubuntu 18.04 LTS running on the same machine as one described in Sect. 4. In these experiments, we answer the following fundamental questions: 1) how is the runtime overhead of our approach?, 2) how does our approach mitigate performance degradation after software rejuvenation?, and 3) how can our approach recover from failures?

5.4.1 Runtime Overhead

To show our prototype’s runtime overhead, we measure the throughput of both vanilla and our memcached under the YCSB benchmark. Like the MyRocks case, we ran all the types of the workload of YCSB, a, b, c, d, e, and f.

The result is shown in Fig. 8. The figure reveals that the runtime overhead is less than 1%. The overhead that comes from checksum calculation and permission changes is negligible in all the cases. We note that our mechanism adds three checksums (12 bytes) and pointer (8 bytes) for each KV. Although the space overhead of our mechanism is bigger relatively if KVs are smaller, KVs’ size on the modern workloads is more than 150 bytes [11], [39], which means our space overhead is less than 1%.

5.4.2 Throughput across Reboot-Based Recovery

To demonstrate that our approach mitigates performance degradation after reboot-based recovery, we measure the throughput of the prototype under its restart. We prepare an intentional workload that models a database service consisting of a fast cache and backing store. The workload stores 1M items, each of which is 128 bytes, and then accesses them randomly. When cache misses occur, it inserts a 1 second latency to simulate backing store accesses. We restart our prototype after the read access phase runs for 10 hours. For comparison, we do the same thing using a vanilla memcached.

Figure 9 shows the result. From the figure, we can see that our prototype successfully mitigates the performance degradation. The restart of the vanilla memcached takes 3.3 hours to warm the cache up while the time for cache warming of ours is milli-second level. The prototype’s downtime is for salvaging slabs on shared memory and reconstructing data structures such as hash tables. Since this time is less than 60 seconds, the prototype produces full performance just after its rejuvenation.

5.4.3 Fault Injection

To demonstrate our mechanism protects KV pairs against illegal writes, we perform fault injections into the running prototype. We perform a memory write on a random part of the running prototype’s shared memory 100 times. We also observe whether the prototype is running correctly when the faults manifest.

The result is that the prototype successfully stops in a fail-stop manner in all the cases. This means any KV pairs are not corrupted, and thus we can restart the prototype with preserved KV pairs. We note the prototype may fail to restart incorrectly due to writes to all the checksums and writable slabs, but the probability is relatively small.

6. Related Work

Several mechanisms for efficient software rejuvenation have been proposed. MicroReboot [13] achieves fine-grained Java application reboots. To enable a microreboot, the target Java application is divided into small independent software components that become units for a reboot. While MicroReboot is based on software components, our approach exploits characteristics of memory objects; performance related objects (data content objects) and aging related objects (management objects). Our concept is applicable to MicroReboot to reboot software components efficiently.

NVMCached [37] is a variant of memcached tailored to non-volatile memory. To avoid frequent flushes for storing KV pairs in non-volatile memory regions, NVMCached stores checksum of data along with its corresponding metadata so that the data integrity can be verified when necessary. NovelSM [26] is an LSM-tree that constructs block caches on non-volatile memory to mitigate the performance degradation after reboots. Unlike them, our approach allows us to safely restore KV pairs by keeping KV pairs’ pages readonly as much as possible. Also, our approach can preserve KV pairs across its restarts on DRAM-only machines.

Phase-based Reboot [38] shortens the downtime of reboot-based recovery. It takes snapshots at every boot phase, such as the OS kernel boot and service process boot
phases, and reuses them if the next boot has the same execution as the previous boot. Otherworld [20] achieves the reactive rejuvenation of OS kernels by rebooting the kernels without clobbering the state of the running applications. After the kernel crashes and is rebooted, Otherworld [20] restores the application memory spaces, open files, and other resources. Roothammer [28], [29] and ReHype [30] achieve a fast hypervisor rejuvenation by preserving the running VMs in memory while rebooting the hypervisor. Our focus is on the efficient reboot-based recovery of in-memory KVSes.

7. Conclusion

Reboot-based recovery for in-memory DBs is essential to attain high availability of services for periodically and/or reactively recovering from their software bugs’ harmful effects. However, reboot-based recovery causes severe performance degradation due to the elimination of data content objects. This paper presents software mechanisms for efficient reboot-based recovery of in-memory DBs. Our key insight is to decouple data content objects from all the memory objects. We specifically preserve data content objects on memory while refreshing the other objects. Although more data content objects cause longer restart time, the restart time is much shorter than the conventional warm-up time. Our approach treats data items such as DB blocks and KV pairs as data content objects and keeps them on memory across in-memory DB’s restarts in a safe manner. To show the effectiveness of our approach, we implemented our prototypes using two real-world DBs, MyRocks and memcached. The experimental results show that the prototypes significantly mitigate the performance degradation compared to regular reboots, and their runtime overhead is up to 8%.

References

[1] Getting started with myrocks facebook/mysql-5.6 wiki. https://github.com/facebook/mysql-5.6/wiki/Getting-Started-with-MyRocks.
[2] Github. Memcached memory leak issue. https://github.com/memcached/memcached/issues/309.
[3] Memcached. http://memcached.org/.
[4] Memcached - issue no.392. https://code.google.com/archive/p/memcached/issues/392.
[5] Memcached 1.5.18 release notes. https://github.com/memcached/memcached/wiki/ReleaseNotes1518.
[6] Mysql. https://www.mysql.com/.
[7] Mysql bugs. https://bugs.mysql.com/.
[8] Mysql issues: Memory leak with only select statements. https://github.com/facebook/mysql-5.6/issues/774.
[9] Rocksdb: Block cache. https://github.com/facebook/rocksdb/wiki/Block-Cache.
[10] Rocksdb wiki. https://github.com/facebook/rocksdb/wiki.
[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” Proc. 12th ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’12), vol.40, no.1, pp.53–64, June 2012.
[12] A. Bovenzi, J. Alonso, H. Yamada, S. Russo, and K.S Trivedi, “Towards fast os rejuvenation: An experimental evaluation of fast os reboot techniques,” 2013 IEEE 24th Int. Symp. Software Reliability Engineering (ISSRE), pp.61–70, IEEE, 2013.
[13] G. Canda, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot—a technique for cheap recovery,” USENIX Association OSDI ’04: 6th Symposium on Operating Systems Design and Implementation, pp.31–44, 2004.
[14] D. Carra and P. Michiardi, “Memory partitioning and management in memcached,” IEEE Trans. Services Computing, vol.12, no.4, pp.564–576, July–Aug. 2016.
[15] P.M. Chen, W.T. Ng, S. Chandra, C. Aycock, G. Rajamanil, and D. Lowell, “The rio file cache: Surviving operating system crashes,” Proc. 7th ACM Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’96), vol.30, no.5, pp.74–83, Sept. 1996.
[16] B.F Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," Proc. 1st ACM Symposium on Cloud Computing (SoCC ’10), pp.143–154, June 2010.
[17] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “Software aging analysis of the linux operating system,” 2010 IEEE 21st Int. Symp. Software Reliability Engineering, pp.71–80, IEEE, 2010.
[18] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “Software aging and rejuvenation: Where we are and where we are going,” 2011 IEEE Third International Workshop on Software Aging and Rejuvenation, pp.1–6, IEEE, 2011.
[19] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “A survey of software aging and rejuvenation studies,” ACM Journal on Emerging Technologies in Computing Systems (JETC), vol.10, no.1, pp.1–34, Jan. 2014.
[20] A. Depoutovitch and M. Stumm, “Otherworld: Giving applications a chance to survive os kernel crashes,” Proc. 5th European Conference on Computer systems, pp.181–194, April 2010.
[21] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K.S Trivedi, “A methodology for detection and estimation of software aging,” Proc. Ninth Int. Symp. Software Reliability Engineering (Cat. No. 98TB100257), pp.283–292. IEEE, 1998.
[22] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F.U. Haq, and J. Wiener, “Fast database restarts at facebook,” Proc. 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD ’14), pp.541–549, June 2014.
[23] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. “LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache,” 2015 USENIX Annual Technical Conference (USENIX ATC ’15), pp.57–69, 2015.
[24] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton, “Software rejuvenation: Analysis, module and applications,” Proc. IEEE 25th Int’l Symp. Fault-Tolerant Computing (FTCS ’95), pp.381–390, 1995.
[25] Y. Jumonji and H. Yamada, “Efficient software rejuvenation of in-memory key-value storages,” 2017 IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW), pp.280–285, IEEE, 2017.
[26] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Recession: Isms for nonvolatile memory with novelism,” 2018 USENIX Annual Technical Conference (USENIX ATC ’18), pp.993–1005, July 2018.
[27] K. Kourai, “Cachemind: Fast performance recovery using a virtual machine monitor,” 2010 Int. Conf. Dependable Systems and Networks Workshops (DSN-W), pp.36–92, IEEE, 2010.
[28] K. Kourai and S. Chiba, “A fast rejuvenation technique for server consolidation with virtual machines,” 37th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN’07), pp.245–255, IEEE, 2007.
[29] K. Kourai and S. Chiba, “Fast software rejuvenation of virtual machine monitors,” IEEE Trans. Dependable and Secure Computing (TDSC), vol.8, no.6, pp.839–851, Nov.–Dec. 2010.
[30] M. Le and Y. Tamir, “Rehype: Enabling vm survival across hypervisor failures,” 7th ACM Int’l Conf. on Virtual Execution Environments (VEE’11), vol.46, no.7, pp.63–74, July 2011.
[31] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J.
Krueger, and M. Grund, “High-performance transaction processing in sap hana,” IEEE Data Eng. Bull., vol.36, no.2, pp.28–33, 2013.

[32] Y. Matsunobu, “Myrocks: A space and write-optimized mysql database,” https://code.fb.com/core-data/myrocks-a-space-and-write-optimized-mysql-database/.

[33] W.T. Ng and P.M. Chen, “The systematic improvement of fault tolerance in the rio file cache,” Proc. 29th Symposium on Fault-Tolerant Computing (FTCS ’99), pp.76–83, 1999.

[34] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” Acta Informatica, vol.33, no.4, pp.351–385, June 1996.

[35] K. Terada and H. Yamada, “Dwarf: Shortening downtime of reboot-based kernel updates,” 2016 12th European Dependable Computing Conference (EDCC), pp.208–217, IEEE, 2016.

[36] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, “Espresso: Brewing java for more non-volatility with non-volatile memory,” Proc. Twenty-Third Int. Conf. Architectural Support for Programming Languages and Operating Systems, vol.53, no.2, pp.70–83, Feb. 2018.

[37] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang, ”Nvmcached: An nvm-based key-value cache,” Proc. 7th ACM SIGOPS Asia-Pacific Workshop on Systems, pp.1–7, Aug. 2016.

[38] K. Yamakita, H. Yamada, and K. Kono, “Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery,” 2011 IEEE/IFIP 41st Int. Conf. Dependable Systems & Networks (DSN), pp.169–180, IEEE, 2011.

[39] J. Yang, Y. Yue, and K.V. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at twitter,” Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’20), pp.191–208, 2020.

Yuto Jumonji received his B.E. and M.E. degrees from Tokyo University of Agriculture and Technology in 2017 and 2019. He is currently a Ph.D. student in the Department of Information and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and dependable computing.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, and cloud computing. He is a member of IEEE/CS, ACM, and USENIX.