Check-In: In-Storage Checkpointing for Key-Value Store System Leveraging Flash-Based SSDs

Joohyeong Yoon[†§], Won Seob Jeong[†§], and Won Woo Ro[†]

[†]School of Electrical and Electronic Engineering, Yonsei University
[§]Memory Division, Samsung Electronics Co.
{jooh.yoon, ws.jeong, wro}@yonsei.ac.kr

***Abstract*—Persistent key-value store supports journaling and checkpointing to maintain data consistency and to prevent data loss. However, conventional data consistency mechanisms are not suitable for efficient management of flash memories in SSDs due to that they write the same data twice and induce redundant flash operations. As a result, query processing is delayed by heavy traffics during checkpointing. The checkpointing accompanies many write operations by nature, and a write operation consumes severe time and energy in SSDs; worse, it can introduce the write amplification problem and shorten the lifetime of the flash memory. In this paper, we propose an in-storage checkpointing mechanism, named Check-In, based on the cooperation between the storage engine of a host and the flash translation layer (FTL) of an SSD. Compared to the existing mechanism, our proposed mechanism reduces the tail latency due to checkpointing by 92.1 % and reduces the number of duplicate writes by 94.3 %. Overall, the average throughput and latency are improved by 8.1 % and 10.2 %, respectively.**

***Index Terms*—checkpointing, in-storage processing, key-value store, solid state drives, flash memory**

I. Introduction

Persistent key-value store [2], [5], [15], [21] is widely applied to diverse data processing services, including cloud services, social networking, online shopping [23], and messaging [29]. The key-value store has no strict structure and allows variable-length of key-value for data store and retrieval. The key-value store becomes an important component in big data environments due to its flexible and horizontally scalable characteristics, compared to the traditional relational databases [1], [3], [49], [51].

Meanwhile, the storage landscape is also changed. Flash-based solid state drives (SSDs) become popular as a storage device for offering decreased latency and increased bandwidth compared to conventional disk drives. Thanks to the emergence of cost-effective and high-capacity SSDs, SSD's market share is steadily increasing in many enterprise storage systems. With the interest of incorporating flash memory into databases, various types of flash-based key-value stores have been researched [12], [21], [22], [32], [42]–[44], [47], [55].

The persistent key-value store system, that uses SSDs for storing data, generally leverages journaling and checkpointing to ensure data consistency and to prevent data loss in case of power failure or system crash [3], [9]. These mechanisms are handled by the storage engine in the database management system (DBMS). The storage engine writes journal logs to the underlying SSD and creates multiple checkpoints periodically. The data structure can be restored from the last checkpoint and journal logs if there is a power failure or a system crash.

However, this data consistency model that uses both journaling and checkpointing exhibits the following problems. The first problem is write amplification, which writes the same data twice, rewriting the data written by journaling at the checkpoint. The write amplification worsen the energy consumption, degrades the performance [8], [14], [17], [18], and shorten the lifetime of SSDs [45], [54]. As flash memory has a limited program/erase cycle (P/E cycle) that determines its lifetime, increasing the number of writes can unfortunately shorten the lifetime of flash memory. Especially, the key-value store offers a small set of operations for data storage and retrieval [10]. So, frequent updates of small key-value data generate lots of invalid flash pages, which can invoke garbage collection (GC), because the small key-value data are typically stored misaligned with the mapping unit of the storage device. The frequent data migration induced by GC would amplify the write operation and degrade performance due to the random access characteristic of SSDs [48]. The second problem is the long latency of query processing due to checkpointing. Excessive I/O traffics during checkpointing can worsen the query processing latency in the persistent key-value store system.

We analyze the checkpointing overhead in the persistent key-value store system. Our experimental results reveal that 1) the number of flash operations in an SSD is increased by the I/O requests due to checkpointing and 2) the query processing latency during checkpointing is increased dramatically; the results of the initial study are shown in Figure 3. The amount of total I/O requests is increased by 2.98 times when data access distribution is uniform and by 1.91 times when it is Zipfian, compared to the amount of data in write queries. Moreover, these I/O requests raise the number of flash operations by 7.9 times at uniform distribution and by 4.7 times at Zipfian distribution due to internal write amplification of the SSD. The results also show that the query processing latency during checkpointing is increased by 4 times for read queries and by 21 times for write queries compared to the average query processing latency. The dramatic latency increase during checkpointing is caused by the burst flash operations in an SSD. In summary, checkpointing induces

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00063

the write amplification and increases the imbalance among query processing. The write amplification and the quality of service (QoS) are critical factors in database applications, so the redundant writes and the tail latency due to checkpointing should be reduced.

Based on these observations, we propose an in-storage checkpointing mechanism, named *Check-In*, that the flash translation layer (FTL) of an SSD cooperates with a storage engine of a host to reduce the write amplification and improve QoS in query processing. Check-In offloads checkpointing to the FTL and the mechanism includes the algorithm that aligns journal logs to the FTL mapping unit, called sector-aligned journaling, to reduce the internal write amplification. The FTL creates a checkpoint by remapping the journal logs to the checkpoint. Conventional FTLs already adopt a log-like write method and an indirect mapping [13], [19], [26], [54] between a logical block address and a physical block address because of: 1) the difference between the operation unit of block device interface and the physical page of flash memory and 2) the out-of-place update characteristics of flash memory. By leveraging these characteristics, our FTL updates the mapping information for referencing by the checkpoint logically, instead of physically redundant writes during checkpointing. In this case, aligning the journal logs to the FTL mapping unit is critical for improving the remapping efficiency since the aligned logs can reduce the generation of invalid pages when they are reused for a subsequent checkpointing.

Check-In decreases the number of redundant writes by improving remapping efficiency. Also, our scheme can improve unbalanced query processing latency due to shortening checkpointing time. In addition, distributed parallelism can be exploited; the storage engine can simultaneously process other queries while the FTL creates checkpoints. Our simulation-based studies show that Check-In can improve performance while ensuring the fair provision of services. When Check-In is applied, the tail query latency is reduced by more than 92.1 % during the checkpointing, compared to the baseline. It also reduces the number of duplicate writes by 94.3 %, on average. Overall, query throughput is improved by 8.1 % and query latency is reduced by 10.2 %, on average, compared to the baseline.

Our main contributions can be summarized as follows:

- Offloading the checkpointing to the storage device level avoids unnecessary data transfer between the storage device and main memory with low host CPU overhead. In addition, the proposed sector-aligned journaling approach improves remapping efficiency when journal logs are reused for the next checkpoint in the key-value store system.
- The proposed architecture improves the lifetime of flash memory by reducing the number of duplicate write operations due to checkpoints and by reducing the generation of invalid pages through the high remapping efficiency of the sector-aligned journaling. In addition, our scheme improves the QoS in query processing by shortening checkpointing time.
- The proposed Check-In system works well in lots of small write situations. This is especially helpful in the key-value store which has small key-value data mostly. Moreover, our approach can be applied to other storage systems that use journaling and checkpointing (e.g., a file system) since relatively large data also can be processed effectively.
- Check-In, which cooperates between the host system and the device, can keep the typical system architecture and the firmware structure of the storage device. This is due to exploiting the level of indirection already provided by the FTL to let data stay physically in place, but be referenced by the checkpoint logically. The host system is only required to support reformatting the journal log.

Although there are studies on flash-conscious key-value stores [21], [22], [42]–[44], [47] to reduce write amplification, these approaches can add high remapping overheads on the host CPU. To reduce the CPU overhead, the techniques offloading the remapping operation to the storage device have been studied [7], [24], [28], [36]. These techniques can reduce the number of redundant writes or eliminate remapping overhead on the host CPU. However, prior studies have low remapping efficiency because they cannot utilize the key-value store characteristics, which has variable and mostly small key-value data. There also have been studies on key-value SSDs with a key-addressable interface [12], [32], [55], as well as transactional SSDs [46], [50], [52]. These studies are quite innovative, but offloading all key-value stores that require a lot of resources is costly inefficient. Rather, it can be more effective if that amount of memory resources is utilized for the main memory. Therefore, the storage device that is supporting journaling and checkpointing can be an alternative solution considering resource utilization.

The rest of this paper is organized as follows. Section II introduces conventional journaling and checkpointing and set the design goal which are suitable for SSD characteristics. Section III presents our in-storage checkpointing for reducing the write amplification and improving QoS in query processing. Section IV analyzes the experimental results on our key-value store system. Section V discusses related work. Finally, conclusion is given in Section VI.

II. Background and Motivation

In this section, we introduce the journaling and checkpointing of database systems and describe their key challenges based on an analysis of checkpointing overhead. In addition, we define the design goals of our journaling and checkpointing leveraging the characteristics of flash-based SSDs.

A. Conventional Journaling and Checkpointing

A database engine should provide reliability to prevent data loss. Unfortunately, system crashes or power loss during major updates can lead to database corruption, which is called as the crash-consistency problem [9]. The most prominent solution to the crash-consistency is data journaling, also known as write-ahead logging. When some data are updated on storage

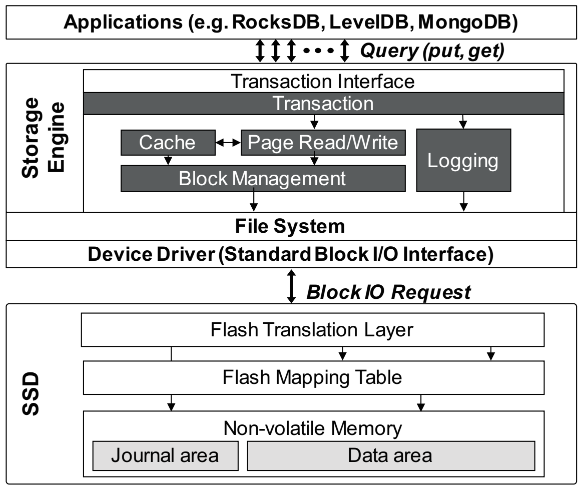


Fig. 1. Conventional database system architecture.

devices, the data are stored in a different logical area before the existing data structure is overwritten. The different logical area is called a journal area, and the written data are called journal logs. The data structure can be guaranteed by restoring to a previous version even if there is a crash-consistency while updating the data structure. Conventional storage engines for key-value stores, such as MMAPv1 and WiredTiger [3], provide checkpointing as well as journaling to provide data consistency. In the engine, the data structure is restored from the last checkpoint first when recovering from the crash-consistency. If a system process is terminated unexpectedly between checkpoints, the data structure is restored based on the journal logs that are written after the last checkpoint.

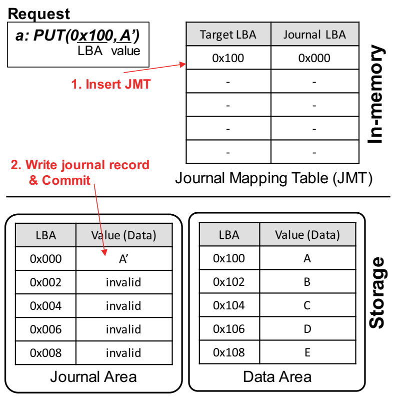
Figure 1 shows a conventional database system architecture that uses SSDs for storing data. The storage engine is the first layer to receive queries from applications and is the key component of the DBMS. The storage engine contains the block management path for normal operations and the logging path for journaling. The block management engine typically manages data in memory. To ensure data consistency and durability, the logging engine stores journal logs sequentially in the journal area on the storage device. This scheme is known as journal synchronization. In addition to this data journaling, the logging engine also supports checkpointing. Checkpointing is the writing of the latest version of data from the journal logs to the final destination of the data area. A typical checkpointing collects journal logs until the next checkpoint and periodically creates checkpoints, which are the reference point for recovery.

B. Case Study for Journaling and Checkpointing

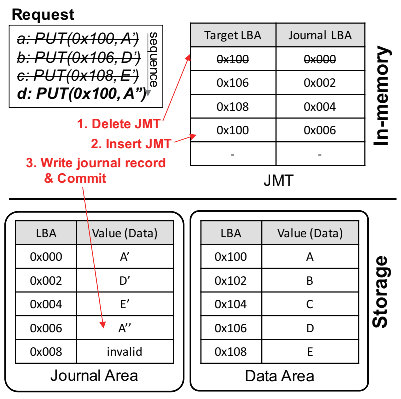
Figure 2 shows data changes in the main memory and the storage device in the journaling and checkpointing scenario. The storage space is logically divided into the journal and data areas. The example assumes that the logical address 0x0 through 0x9 is the journal area and the starting address of the data area is 0x100 in the SSD. A primitive request from applications is expressed in the form of key and value like $PUT(key, value)$. For simplicity, we assume that all keys have already been translated to the target addresses which uses a logical block address (LBA). So, each request is expressed in the form of $PUT(target\ LBA, value)$. Data from key A to E are stored from LBA 0x100 to LBA 0x108. Assume that each data size is 1 KB. Since one logical block size is 512 bytes, the start address of each data is incremented by 0x2. The data for each key is represented by the name of the key, and prime symbols are added according to the version. For example, the data for key A is expressed as A, A', and A'' when further updates are applied. To manage journal logs, a journal mapping table (JMT) is used to store the mapping information between a target LBA and a journal LBA. Elements in the JMT are stored sequentially using the write-ahead log method.

In this situation, Figure 2(a) shows that an update request for key A comes in. New data is not directly updated to the LBA 0x100 in the data area but a new journal log is stored to the LBA 0x0 which is in the journal area. The mapping information of the target location and the journal location corresponding to key A is updated in the JMT. Next, the journal log in the main memory is written to the SSD and committed when the operation completes safely. Subsequent write requests for D' and E' are also recorded in the journal area. Figure 2(b) shows a further scenario. As shown in the example, if an update request for key A, which already exists in the JMT, is updated again with A'' and this log must be stored to the next location of the accumulated journal logs. New mapping information is updated to the JMT without modifying the existing journal log due to the write-ahead log method. Then, the storage engine sends a block I/O request to write a new journal log to the SSD.

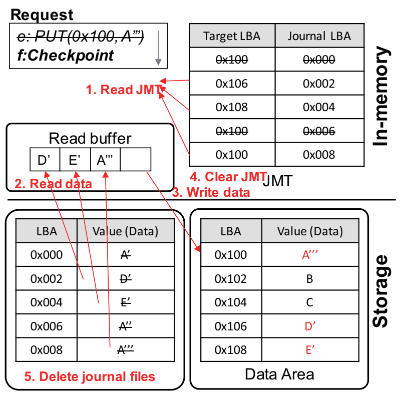
If checkpointing is triggered by a user or a system, the latest data in the journal area should be checkpointed to the data area. Figure 2(c) shows the scenario that the checkpointing is triggered after three updates for key A have been completed. The latest version of data for key A, A''' exists at LBA 0x8 last written to the journal area. D' and E' should also be stored to the target location in the data area. For these operations, the storage engine first reads the JMT and checks for the key information that needs to be checkpointed. To read the latest version of data, the storage engine requests journal data to the SSD. In order to temporarily store the read data, a read buffer should be allocated in memory. Because the latest data read from the journal area should be sent back to the data area, the storage engine requests the write operations to the SSD. All the latest data, recorded in the journal area, are checkpointed to the data area, and related metadata information is also updated additionally. If the checkpointing is safely terminated, the JMT is cleared. After that, used journal data are also flushed because they are no longer needed. The service requests for clients should be supported during checkpointing. Before checkpointing, new journal area and JMT are already built as an alternative, so journaling for other requests can be done without blocking.



(a) Request a: an update request for key A.

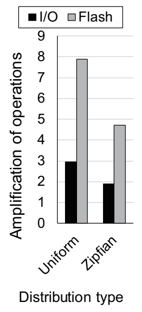


(b) Request d: an update request comes in for a key that already exists in JMT.

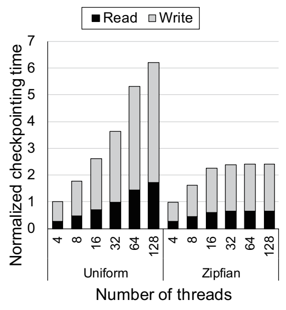


(c) Request f: a checkpointing request.

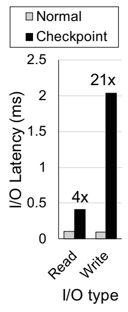
Fig. 2. Illustration of the journaling and checkpointing on the example request sequence. Each figure describes a snapshot of when a particular request is processed during successive requests a through f.



(a) Amplification of operations.



(b) Checkpointing time.



(c) Latency.

Fig. 3. Analysis of checkpointing overhead. (a) The checkpointing amplifies the amount of I/O and flash operations against the amount of data in write queries. (b) The creation time of a checkpoint increases as the number of threads increases. (c) During checkpointing, query processing latency is 4 times longer than normal (no checkpointing) for reading and 21 times longer than normal for writing.

C. Analysis of Checkpointing Overhead

As mentioned previously, the number of I/O accesses during a checkpointing is inevitably higher than when there is no checkpointing. Checkpointing can be required even though storage access for query processing is heavy. So, checkpointing negatively affects the service latency of query processing.

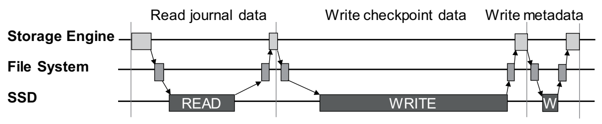
Since a few bytes of data are updated frequently in key-value stores [10], this paper focuses on updates of small key-value items, such as 512 bytes or less. In the case of updating small data, the storage engine reads multiple small amounts of the journal logs stored in the storage device and move them back to the data area in the storage device. These small random accesses degrade the SSD performance [8], [17], [18], [48].

We experimented with our key-value store system simulator, which is based on gem5 [11] and SimpleSSD [25], [33]. The simulator also models a key-value store system to support various configurations and analyze internal operations. During experiments, the number of threads is explored from 4 to 128, which represents the number of applications that access the key-value store simultaneously. As the number of threads increase, storage throughput increases, which also increases the number of journal logs to be checkpointed. Two types of YCSB [6] access patterns are used: uniform and Zipfian distributions. More detailed system configuration is presented in Table I.

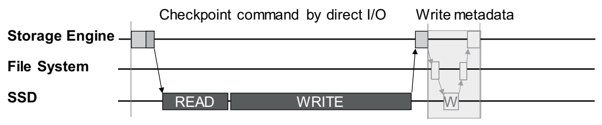
Figure 3(a) shows the amplification of I/O and flash operations due to checkpointing. The amount of total I/O requests is increased by 2.98 times when data access distribution is uniform and by 1.91 times when it is Zipfian, compared to the amount of data in write queries. This amplification increases the number of flash operations by 7.9 times at uniform distribution and by 4.7 times at Zipfian distribution due to internal write amplification of SSD. In particular, the write amplification causes frequent GCs, which in turn decreases the lifetime of flash memory as well as the performance. Therefore, eliminating unnecessary data duplication can improve both performance and lifetime.

Figure 3(b) shows the normalized checkpointing time. The graph reveals that the checkpointing time tends to increase as the number of threads increases. The increasing slope of checkpointing time is greater when using a uniform distribution than when using a Zipfian distribution. This is because as the journal log grows, the number of recent versions in the uniform distribution continues to increase, but the number of latest versions in the Zipfian distribution saturates at a certain level. When 128 threads are executed, the ratio of the latest version to the uniform distribution is 5.02 times greater than the ratio of the latest version in the Zipfian distribution, and the number of I/O access is increased by that much.

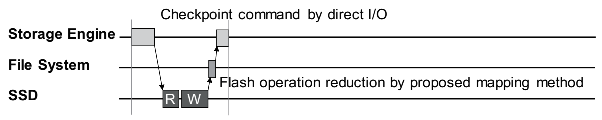
The performance evaluation on a real machine shows that query processing is significantly delayed during checkpointing. Figure 3(c) shows the difference between I/O latency during



(a) Conventional checkpointing.



(b) In-storage checkpointing with conventional FTL.


Checkpoint command by direct I/O
Storage Engine
File System
SSD
R
W
Flash operation reduction by proposed mapping method

(c) In-storage checkpointing with the proposed storage engine-aware FTL.

Fig. 4. Timeline of checkpointing. The overhead due to checkpointing can be reduced by offloading the function from the storage engine to the storage device. In addition, storage engine-aware FTL can reduce the number of flash operations.

journal flush (or checkpointing) and average I/O latency when processing queries through the MMAPv1 storage engine in a system with an Intel 750 SSD. Therefore, to improve the quality of service, the checkpointing time should be reduced. A simple way to reduce the checkpointing time is to reduce the interval between checkpoints. However, a shorter checkpoint interval increases the checkpointing time per total operation time. Because updates to the same key in the key-value store are very frequent, short checkpoint intervals can increase duplicate writes as well as total operation time.

On the other hand, the interval between checkpointing can be increased to reduce the checkpointing time per total operating time. A longer checkpoint interval increases the amount of data to be checkpointed at one time but takes benefit from the overwriting data to the same location due to the lazy update. The data reduced by this redundancy can reduce checkpointing time per total operation time. However, in order to increase the checkpoint interval, a lot of space is required due to the journal area, which is limited by the system. In addition, long checkpoint intervals can significantly increase recovery time from power outages or system failures. Therefore, a proper trade-off between checkpoint interval and the system configuration is required.

D. Design Goals of Journaling and Checkpointing Leveraging SSD Characteristics

During checkpointing, transferring data from a storage device to the main memory and transferring that data back to the storage device cause redundant overhead. As shown in Figure 4, this overhead can be removed by offloading checkpoint-related functions from the host system to the storage device. Figure 4(a) shows an example of a conventional checkpointing. During checkpointing, journal logs are copied from the storage device into memory and the latest data among them are written back to the storage device. The updated metadata is then written to the storage device. Reading and writing data through the block interface between main memory and the storage device can be quite time-consuming. However, if the checkpointing is offloaded to the storage device, the checkpointing time can be improved as shown in Figure 4(b). Writing metadata can also be hidden.

In addition to offloading, if the FTL is aware of the journaling in a host system, it can reduce the number of reads and writes to flash memory. Figure 4(c) shows the timing diagram expected when the storage engine-aware FTL is applied with the in-storage checkpointing. Even if all reads and writes are not removed, most of the unnecessary flash operation can be reduced, which greatly reduces the checkpointing overhead.

SSD features must be fully utilized to implement the in-storage checkpointing and the storage engine-aware FTL. An SSD has multiple embedded processors that can execute firmware, so-called FTL. To improve the performance and reliability of data access in an SSD, the FTL processes host requests according to flash memory characteristics. Due to the out-of-place update characteristic of flash memory, the FTL maps logical pages to physical pages and updates a mapping table in a log-structured way. We exploit this log-structured FTL in order to remove the redundant writes by journaling and checkpointing. If checkpointing on the FTL is replaced by copy-on-write (CoW), which reuses already written journal logs to checkpoints, it can reduce write amplification.

In order to increase the efficiency of CoW, the reusability of data is very important. Small data in the key-value store is not suitable for the typical FTL mapping unit, which can degrade the reusability of data. To improve the reusability of the journal logs at the checkpoint, the key-value data must be aligned to the FTL mapping unit during journaling. The alignment of the journal logs can be supported by the storage engine considering resource utilization between memory and storage devices.

In addition, NVMe, a new standard interface for high-end SSDs, is block-oriented but allows special-purpose requests by allowing vendor-specific extensions [4]. This feature helps to communicate the workload between the storage engine and the FTL. Detailed techniques are further presented in Section III.

III. In-Storage Checkpointing

This section presents our Check-In architecture consisting of two components: the Check-In engine and the Check-In SSD. Also, we introduce mapping table management in the Check-In SSD and sector-aligned journaling in the Check-In engine.

A. Check-In Architecture

Figure 5 shows our Check-In architecture. Each thread in an application requests various types of queries to the Check-In engine. The Check-In engine performs the storage management tasks such as key-value mapping, journaling, and checkpointing according to incoming requests. When these

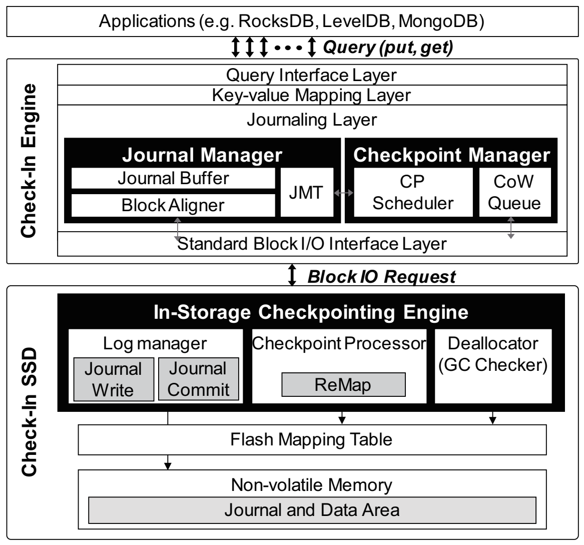


Fig. 5. Check-In Architecture. The Check-In engine consists of a journal manager and a checkpoint manager. The Check-In SSD includes an in-storage checkpointing engine to manage the data journaling and checkpointing.

tasks are conducted, it accesses system memory or the Check-In SSD as needed. The communication between the Check-In engine and the Check-In SSD is via a standard block I/O interface.

The Check-In engine consists of the following four layers: a query interface layer, a key-value mapping layer, a journaling layer, and a standard block I/O interface layer. Among them, the journaling layer controls journaling and checkpointing for data reliability and consistency. A common way to manage journal files is to add new journal records to the journal head, such as log-structured data [9]. Therefore, our journaling mechanism is designed based on log-structured data. It also includes journal data buffering and caching, which is supported to improve performance on several storage engines.

The controller in the Check-In SSD adds an in-storage checkpointing engine (ISCE) in the existing SSD architecture. ISCE consists of a log manager, a checkpoint processor, and a deallocator. The log manager writes the journal logs sent by the Check-In engine's journal manager and informs the Check-In engine that logs have been committed successfully. The log manager also periodically updates the metadata, which will be used later for recovery operations after the last checkpoint. The checkpoint processor creates checkpoints by inserting or modifying elements in the flash mapping table based on the metadata. It reads and writes data via DMA engines separately from the embedded processor as needed. The deallocator frees elements of flash mapping table when deleting journal logs that have already been checkpointed, and determines whether to invoke GC depending on the idle status of the Check-In SSD.

B. Overview of Checkpointing

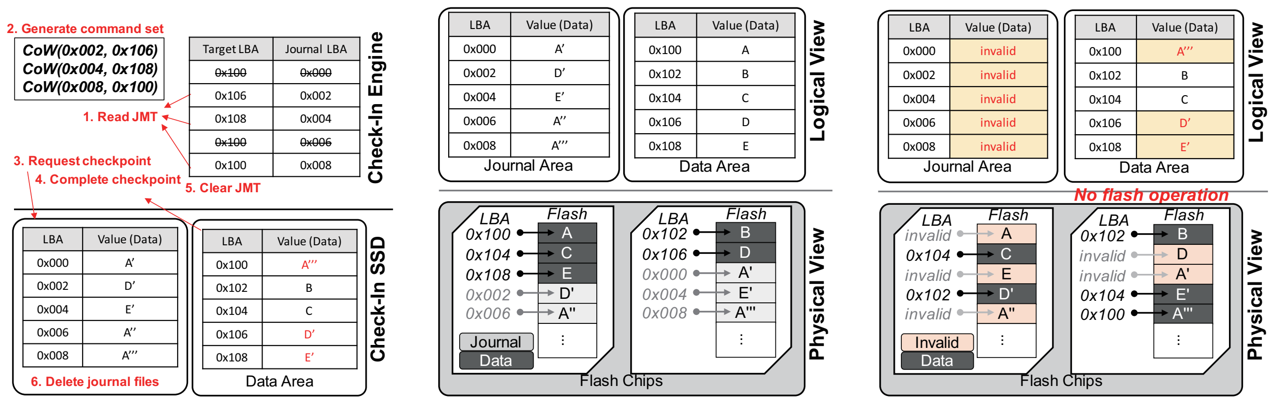
Figure 6(a) shows the operation of the proposed checkpointing mechanism. When checkpointing is triggered, the Check-In engine first reads multiple sets of a target LBA and a journal LBA from a JMT. Based on the information, the Check-In engine creates a set of CoW commands for the Check-In SSD. The CoW command is that physically stored data are shared between the journal area and the data area without copying the data in the journal area to the data area. The checkpointing is then offloaded to the Check-In SSD via the block interface. After checkpointing, the Check-In SSD notifies the Check-In engine that a checkpoint is created. The Check-In engine then clears the JMT entries used to create the checkpoint and sends a command to the Check-In SSD to delete the associated journal logs.

Figure 6(b) and Figure 6(c) show the status before and after checkpointing in the Check-In SSD. The logical view is the status of the stored data viewed by the Check-In engine, and the physical view is the mapping status between logical and physical pages in the Check-In SSD. When the checkpointing is completed, the data existed in the data area are replaced with the latest data in the journal area. After updating the data mapping, the data in the journal area are invalidated. However, the physical pages of flash memory have no change before or after checkpointing. Instead, only the mapping information in the flash mapping table is updated. The physical pages of the latest version are mapped to the logical pages corresponding to the data area.

C. Checkpointing Request and Function Offloading

Our solution to the checkpointing overhead is to offload the checkpoint-related functions from the storage engine to the storage device. Offloading checkpointing requires co-work on both the Check-In engine and the Check-In SSD. The execution code for offloading is sent to the Check-In SSD only once before the first execution. The Check-In engine then requests to the Check-In SSD with only the command set including the source address and the destination address without data transfer. Recent efficient block interfaces allow for special commands, also called vendor-specific commands. In short, the Check-In engine reads journal mapping information in the memory cache before checkpointing and sends the copy request commands to the Check-In SSD. In this paper, the name of the commands is called the CoW command because the checkpointing in the SSD behaves like CoW, which is one of the deduplication methods.

The CoW request command removes the data transfer from the communication of traditional commands, so it reduces the occupation time of the block interface. However, the number of CoW commands also increases as the size of the journal logs that need to be checkpointed increases. The latency to process queries may increase by limited command queue depth. After checkpointing is finished, as shown in Figure 4(b), the Check-In engine will send metadata to write to the storage device. If the checkpoint request command including this metadata is transmitted in advance, the Check-In SSD can decode



(a) Overview. (b) Before checkpointing. (c) After checkpointing.

Fig. 6. Operation of Check-In. (a) describes operational steps of checkpointing. (b) and (c) show the contents of the internal SSD data before and after checkpointing in logical view and physical view.

it and operate multiple CoWs. As a result, the number of commands can be greatly reduced, and the command overhead is reduced to a negligible level. In addition, multiple CoW operations can be separated into consecutive read operations and consecutive write operations, which improve the efficiency of flash memory access during checkpointing.

By offloading the checkpointing, it is possible to eliminate the data transfer overhead between a memory and a storage. In addition, CPUs and SSDs can process checkpointing in parallel, which improves efficiency and performance. However, only offloading cannot be a complete solution since the overhead of reading and writing flash memory during checkpointing still exists. Due to the rapid development of the block interface, the performance of the SSD is greatly affected by the bandwidth of the flash memory. Therefore, it is very important how the checkpointing should be handled on the SSD, which is discussed in detail in the next section.

D. Remapping

The write-amplification of flash memory due to the checkpointing can be reduced by modifying the page mapping method of FTL. Among the various mapping methods, sub-page mapping is widely used [31], [37]. The sub-page mapping uses a logical page size smaller than the physical page size and manages flash pages in the smaller granular unit.

Our approach applies the sub-page mapping and adjusts the mapping unit to a journal log size formatted in the Check-In engine. For example, when the journal log size is 1 KB, the FTL uses 1 KB mapping and the logical page size is 1 KB. If the journal log size is the same as mapping unit, redundant read-modify-write operations can be eliminated. If 4 KB mapping is used and only 1 KB of a 4 KB page needs to be updated, the FTL reads the old 4 KB page, replaces only the new 1 KB, and writes the modified page (read-modify-write operation). If 1 KB mapping is used, the FTL simply writes a new 1 KB page.

The correlation between the journal log size and the mapping unit is more important in the checkpointing. The logs recorded by journaling contain the data to be updated and the data is rewritten during checkpointing. If the log size is aligned to the mapping unit, the rewriting problem can be easily solved by the CoW operation. The FTL generally maps one logical page and one physical page but it is also possible to map multiple logical pages to one physical page. The FTL of the Check-In SSD maps the same physical page to both journal and data areas. In this case, redundant writes can be removed and only the mapping information is updated.

Algorithm 1 shows the pseudo-code to create a checkpoint from journal logs running on the Check-In SSD. During checkpointing, the FTL updates the latest data for each key in the JMT. In the table, journal logs with the flag *NEW* are recorded by journaling, and the flag is changed to *OLD* when logs with the same key are subsequently updated. If the flag is not *OLD*, the FTL finds the PPN mapped to the journal LPN and maps it to the target LPN at the checkpoint.

Algorithm 1: Checkpointing on FTL.

```
1:  function Checkpointing()
2:      for it ← JournalMap.begin() to JournalMap.end() do
3:          if it.flag == OLD then
4:              continue();
5:          end if
6:          else
7:              source ← it.journal_lpn;
8:              destination ← it.target_lpn;
9:              ppn ← FlashMapTable.search(source);
10:             FlashMapTable.insert(ppn, destination);
11:         end if
12:     end for
13:     FlashMapTable.update();
14: end function
```

In the Check-In SSD, after all the information of the JMT is checkpointed, the modified flash mapping table is

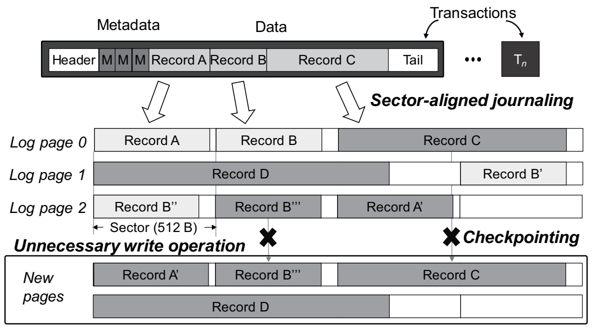


Fig. 7. Proposed sector-aligned journaling scheme.

generally updated to flash memories. However, if the amount of modified information is small, the flash mapping table is not updated immediately. It is written to flash memory when it is accumulated enough to allow parallel processing of the flash memory. This method is based on the commercial features of the SSD: the sudden power-off recovery (SPOR). The internal buffer in the SSD ensures data durability by using power capacitors in the device.

E. Sector-Aligned Journaling

Key-value stores have the advantage of managing data in various formats, without being restricted to any fixed formats. To support the key-value stores, the FTL must manage the value size different from the logical page size in the storage device. If the value size is larger than the logical page size, it can be handled by dividing the value into multiple logical pages or mapping the value to a larger mapping unit. On the other hand, if the value size is smaller than the logical page size, the mapping unit must be reduced to the smallest value size. This approach requires a significant amount of in-storage memory resources due to the increased amount of mapping information. Recent key-value SSD supports 128 bytes as a minimum value size [32]. The Check-In SSD manages the mapping based on the typical host sector size; 512 bytes.

If the value size is unaligned to the sector size, additional processing is required, as shown Algorithm 2. This algorithm is executed in the block aligner of the journal manager. In function *Update()*, values larger than the mapping size (i.e. 512 bytes) are aligned by $sector\ size \times n$ through compression. Values smaller than 512 bytes are formatted to the size of 128, 256, 384, and 512 bytes, depending on their size. If the reformatted value size is not 512 bytes, their type is changed to *PARTIAL*. The values with the *PARTIAL* type are handled exceptionally. In this case, as shown in function *WriteJournalLogs()*, multiple values are merged and placed together in one sector. The type of merged values is changed to *MERGED*. Then, the journal manager writes the journal logs with these sector-aligned values to the storage device.

Due to merged values, the checkpointing may require additional flash operations. However, these small data can be buffered in the in-storage memory for the next checkpoint. Multiple small data are merged and then programmed in page units of flash memory. So, the exception handling for small data is not a big overhead because the aggregate writes do not require much flash writes.

Algorithm 2: Replacing log size and writing journal logs on storage engine.

```
function Update(request)
    MakeJournalLog($request$);
    if $request.size > MAPPING\_SIZE$ then
        Compress($request$);
        $request.size \leftarrow (request.size/MAPPING\_SIZE + 1) * MAPPING\_SIZE$;
        $request.type \leftarrow FULL$;
    else
        $next\_size \leftarrow MAPPING\_SIZE - MAPPING\_SIZE/4$;
        while $request.size < next\_size$ do
            $next\_size \leftarrow next\_size - MAPPING\_SIZE/4$;
        end while
        $request.size \leftarrow next\_size + MAPPING\_SIZE/4$;
        if $request.size == MAPPING\_SIZE$ then
            $request.type \leftarrow FULL$;
        else
            $request.type \leftarrow PARTIAL$;
        end if
    end if
    JournalLogBuffer.insert($request$);
end function
function WriteJournalLogs()
    for $it \leftarrow$ JournalLogBuffer.begin() to JournalLogBuffer.end() do
        if $it.request.type == MERGED$ then
            $continue()$
        else if $it.request.type == PARTIAL$ then
            $request \leftarrow$ MergePartialLogs($it.request$);
        $requestIO \leftarrow$ TranslateAddress($request$);
        insertIO($requestIO$);
    end for
end function
```

Figure 7 shows the sector-aligned journaling scheme using the mapping unit. The Check-In engine groups the journal logs into a transaction and sends it to the Check-In SSD. Each journal log is sector-aligned and these journal logs are sequentially programmed to the flash page. When a checkpoint request arrives, the FTL updates the mapping information to point to the latest data in the journal logs without creating new pages. With this logic, the proposed scheme can considerably eliminate unnecessary write operations.

F. Garbage Collection

If mapping information is frequently updated by checkpointing, the number of invalid pages in the journal area is increased. This is inevitable for the SSDs that use flash memory with out-of-place update characteristics. However, the proposed Check-In reduces the generation of invalid pages through sector-aligned journaling. In addition, read-modify-write operations due to the checkpointing can be deferred until the next GC. In other words, our Check-In has the advantage of distributing the workloads within the SSD in time. Of course, the GC should be required instantly when the storage capacity is almost full. In general, however, the GC is performed as a background job in an idle state with few workloads. Therefore, it reduces the instantaneous query processing delay during checkpointing.

G. Recovery Scenario

Check-In can manage data recovery on system failures. The FTL of the Check-In SSD can recover the data structure by reading metadata stored in flash memory. The first priority in the recovery scenario is the data recovery policy of the SSD. After the reliability of the storage data is guaranteed, the data structure can be restored by using the checkpoints and the journal logs written from the Check-In engine. The data structure is recovered to the last checkpoint in the same way as the recovery flow on conventional storage engines. Then, journal logs after the last checkpoint are pre-read and cached in the buffer, which can reduce the recovery time by the Check-In engine.

The data recovery of SSD is typically processed by the information contained in the out-of-band (OOB) area existing on the page of flash memory. The Check-In SSD writes the target address (or key) and the version for data recovery to the OOB area. If they are successfully stored, the reliability of the data after checkpointing can be guaranteed without any modification.

H. Overhead

Amount of mapping information. Using a small mapping unit has the overhead of increasing the amount of mapping information. In recent years, the capacity of the in-storage DRAM has been increased, so that it can contain a large number of mapping information. In addition, FTLs have evolved to process the small mapping unit. Sector log [31] and sub-page mapping [34] methods are also used to reduce the amount of mapping information when mapping in a smaller unit than the logical page size. Other studies use caching of mapping information without putting all mapping information on the in-storage DRAM [26]. The Check-In SSD uses page mapping with a small logical page size and applies the mapping cache technique.

Space utilization. Aligning to sector size makes mapping easier, but can reduce the space utilization of the storage device due to unused areas. Our sector-aligning method, which merges multiple values into one sector, can provide better space utilization over the conventional method. In addition, since the write amplification can be greatly reduced, the overall utilization of the storage device can be improved.

IV. Evaluation

This section presents the experimental setup and performance evaluation. We compare the proposed Check-In method to other mechanisms in the key-value store system.

TABLE I
SIMULATED MACHINE CONFIGURATION

DBMS configuration	
Record size	128 - 4096 bytes
Checkpoint interval	60 seconds
Data collection count	10,000,000
Total query count	5,000,000
Host system configuration	
Processor	Intel Xeon E5-2680 V4 (2.4GHz)
Memory	DDR4 (1700 MHz), 16 GBs
PCIe	Gen4, 3940 MB/s
Storage configuration	
Embedded processor	Dual Cortex-R7 core (800 MHz)
Data cache	512 MBs
Memory (map cache)	DDR3 (1066 MHz), 2 GBs
Mapping unit	512 - 4096 bytes
Flash topology	8-channel, 8-die
Flash specification	800 MT/s toggle rate, 16 KB page size, TLC

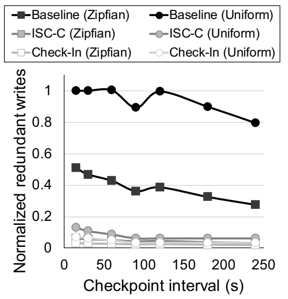
A. Experimental Setup

We design a key-value store system that can simulate journaling and checkpointing. Our simulator is based on gem5 [11], which is widely used for system-wide research with full system simulation. Our simulator consists of the DBMS part and the SSD part.

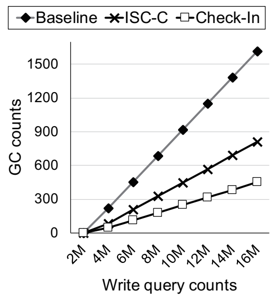
The DBMS contains benchmark applications, a storage engine, and a device driver. For our evaluation, YCSB, one of the most popular benchmarks for NoSQL, is used as the main application. Workload A, workload F, and write-only workload (Workload WO) are used in the experiments. We also use four different patterns of the key-value store that randomly mix various record sizes from 128 to 4096 bytes to evaluate the sector-aligned journaling.

The latest SSD architectures include embedded processors that execute an FTL, NAND flash arrays with multi-level cells, DRAM buffers, and block I/O interfaces using NVMe. We use SimpleSSD [25], [33], a state-of-the-art SSD simulator as a storage device model. The SimpelSSD includes FTL features such as address translation, page mapping, wear-leveling, and GC. So it provides a complete SSD architecture model including host interface, embedded CPU, and DRAM cache with high accuracy. Therefore, the SimpleSSD supporting various structures is very suitable for performance analysis of the key-value store system. We have partially modified the SimpleSSD to implement Check-In SSD. The simulator provides three groups of configurations: DBMS, host system, and SSD. Table I summarizes the configuration of the simulation.

We break down the proposed in-storage checkpointing according to the applied algorithm. The baseline system is a modeled key-value store system, consisting of a storage engine that performs journaling and checkpointing and a typical SSD. We have tested 4 different configurations to show detail performance improvement of Check-In. ISC-A is to use a single page

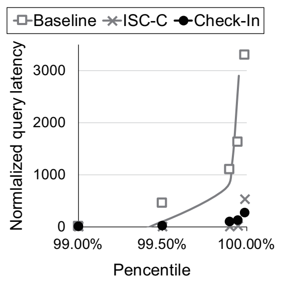


(a) Redundant flash writes.

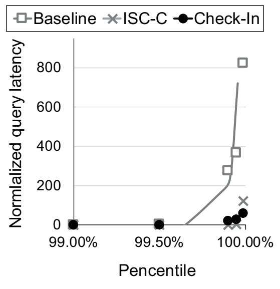


(b) Garbage collections.

Fig. 8. Redundant flash writes and garbage collections. Check-In reduces redundant flash writes by 94.3 % relative to the baseline and reduces GC counts by 74.1 % on average.

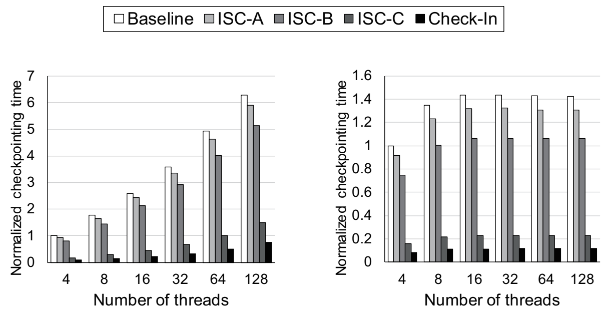


(a) Uniform.



(b) Zipfian.

Fig. 9. Tail latency change by checkpointing overhead. The 99.9th percentile latency is reduced by 92.1 % and 92.4 % compare to the baseline, respectively in uniform and Zipfian distribution.



(a) Uniform. (b) Zipfian.

Fig. 10. Checkpointing time. Compared to the baseline, Check-In reduces the checkpointing time by 92 % and 84 % in uniform and Zipfian distributions, respectively.

(or key) as the copy operation unit in the checkpointing request command method with the conventional mapping method. ISC-B is that the checkpointing request command includes multiple pages (or keys), but the conventional mapping method is still used. ISC-C is that the modified mapping method is applied to FTL and the multiple-page checkpointing request command is used, which has a mechanism similar to near data processing techniques for checkpointing. Check-In is our proposed scheme in which all the modified mapping mechanism and sector-aligned journaling are applied.

- Baseline: Checkpointing by the storage engine.
- ISC-A: In-storage checkpointing by single-CoW commands.
- ISC-B: In-storage checkpointing by multi-CoW commands.
- ISC-C: In-storage checkpointing with remapping.
- Check-In: In-storage checkpointing with remapping and sector-aligned journaling.

B. Write Amplification and Flash Lifetime

The proposed mapping algorithm can reduce the number of flash operations. In addition, reducing the number of flash operations, especially the number of writes and erases for GC, has the effect of extending the flash memory lifetime. We analyze the statistics on flash operations such as write counts and GC counts to evaluate the affect of our mechanism.

Figure 8(a) shows the number of redundant writes on the SSD according to the checkpoint interval. Compared to the baseline and ISC-C, the Check-In SSD reduces the number of redundant writes by 94.3 % and 45.6 % respectively. Fewer write requests to the storage device reduce the number of flash operations, which in turn affects the GC. Figure 8(b) compares the increase in the number of GC invocations by the write query counts. Check-In that supports sector-aligned journaling reduces the GC counts by 74.1 % and 44.8 % compared to the baseline and ISC-C because fewer invalid pages are generated due to the high data reusability of the sector-aligned journaling.

If fewer GC is invoked, fewer P/E cycles are consumed. The lifetime of a flash memory block can be expressed by Equation (1) when $Lifetime_{block}$ is the lifetime of a block, PEC_{max} is the maximum P/E cycle of the block, T_{op} is total operation time of the device, and BEC is current block erase count.

$$Lifetime_{block} = \frac{PEC_{max} \times T_{op}}{BEC} \tag{1}$$

Therefore, the proposed Check-In extends the lifetime of flash memory by 3.86 times compared to the baseline and by 1.81 times compared to ISC-C. Although the number of write I/Os are increased due to the space overhead, Check-In improves the lifetime of flash memory by reducing the flash operations with high remapping efficiency.

C. Tail Latency and Checkpointing Time

We compare tail latency and checkpointing time to evaluate QoS in the key-value store system. In our experiment, checkpointing is triggered at intervals of 60 seconds or when the number of journal files, which are groups of journal logs, accumulates 200 or more. In other words, checkpointing is triggered if the journal area contains more than 2 GB of logs assuming that the journal file size is 100 MB. However, the workloads used in our experiment environment do not fill more than 2 GB of logs before 60 seconds, so in most cases, checkpoints occur every 60 seconds.

Figure 9 shows the tail latency before and after applying our scheme. Check-In reduces the 99.9th percentile latency by 92.1 % and 92.4 %, respectively in uniform and Zipfian distribution compared to the baseline. Likewise, Check-In reduces the 99.99th percentile latency in each distribution by 51.3 % and 50.8 % compared to ISC-C. The result means that each query is processed fairly than the baseline and ISC-C.

These tail latency improvements are closely related to checkpointing time, so we also evaluate checkpointing time. In the behavior of the key-value store model, checkpointing occurs concurrently with normal query processing. In this case, the checkpointing time can be affected by query throughput. To accurately evaluate the checkpointing time, processing of the query request is temporarily locked and enabled again after checkpoints are created.

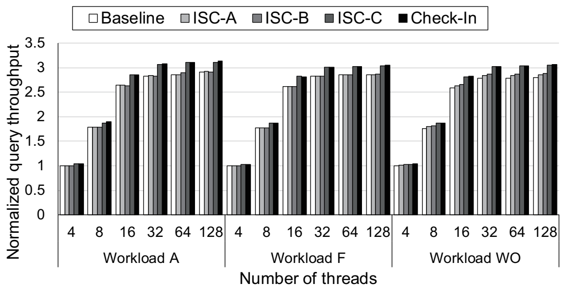
Figure 10 compares the checkpointing time for each configuration according to the number of threads. The proposed in-storage checkpointing shortens the checkpointing time compared to the other mechanisms, especially as the number of threads increases. As the number of threads increases, much more journal data is accumulated at every checkpoint and this increases the amount of data that needs to be checkpointed. As the amount of data to be checkpointed increases, the number of accesses to storage increases instantaneously and leads to greater I/O latency, increasing the checkpointing time. However, even if the amount of accumulated journal data increases, the proposed mechanism has little effect on the checkpointing time comparing to the other mechanisms.

D. Overall Query Throughput and Latency

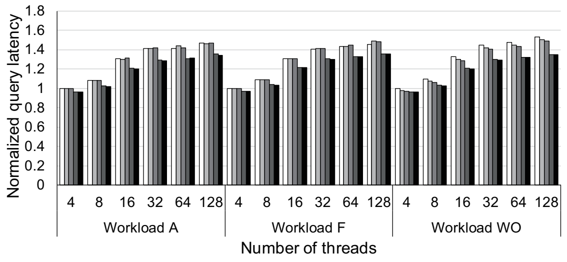
We evaluate overall throughput and latency due to checkpointing overhead in the key-value store system. Workload A, F, and WO, which are write-heavy workloads, are used for evaluation. Workload A contains 50 % of query requests and equal update requests. Workload F contains query requests at a rate of 50 % read and 50 % read-modify-write. Workload WO contains only update requests. The access pattern of the client used in the experiment is Zipfian distribution.

The experiment measures average throughput and latency for three workloads in a key-value store system according to the number of threads, which is the number of queries that the storage engine can process in parallel. As the number of threads increases, the query processing throughput improves, but the query processing latency becomes bigger. This is because the requested query processing may be delayed while other queries are being processed.

Figure 11(a) shows the average query processing throughput as each configuration processes the workloads. The throughput increases as the number of threads increases but saturates at a certain level due to the storage performance constraints. The checkpointing overhead increases the burden on the storage device if the storage performance is constrained. In this case, Check-In can greatly improve the overall performance. Our experimental results show that the average query throughput of Check-In increases as the number of threads increases.

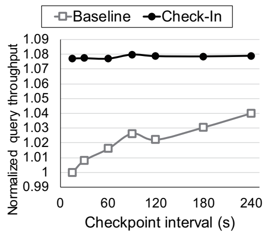


(a) Normalized query throughput compared to the baseline.

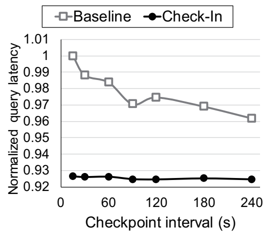


(b) Normalized query latency compared to the baseline.

Fig. 11. Throughput and latency of query processing according to the workload type. Compared to the baseline, the query throughput improved maximum 8.9 %, 7.8 %, and 9.6 % for workloads A, F, and WO, respectively, and the query latency is reduced by maximum 9.3 %, 8.3 %, and 13.7 % for workloads A, F, and WO, respectively.



(a) Query throughput.



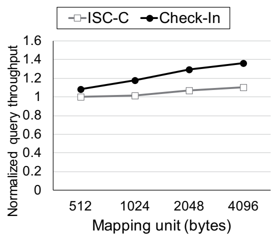
(b) Query latency.

Fig. 12. Sensitivity study according to the checkpoint interval.

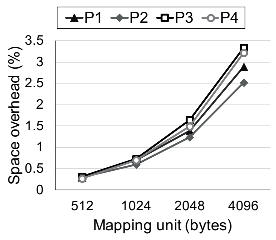
Figure 11(b) shows the average query processing latency. The latency increases as the number of threads increases due to the performance limitations of the storage device. However, Check-In shows the latency improvement of 10.2 % compared to the baseline when 128 threads run. The proposed Check-In improves tail latency, and in addition to this effect, improves overall average latency.

E. Sensitivity Study

We further evaluated the sensitivity of the checkpoint interval and the mapping unit size. In Figure 12, the baseline shows that the query throughput increases and the query latency decreases as the checkpoint interval increases. This is because the update frequency of each key is not balanced and updates to certain keys can be dominant. Meanwhile, Check-In shows



(a) Query throughput.



(b) Space overhead of Check-In over ISC-C.

Fig. 13. Sensitivity study according to the mapping unit size.

better and steady on query latency and throughput regardless of the checkpoint interval.

Figure 13(a) shows the query throughput when the mapping unit size is 512, 1024, 2048, and 4096 bytes and Figure 13(b) presents the space overhead of Check-In versus ISC-C for four different patterns with mixing various record sizes from 128 to 4096 bytes. As the size of the mapping unit increases, the query throughput generally increases because the overhead of processing metadata internally decreases. As a result of the experiment, ISC-C has low data reusability compared to Check-In, so performance improvement by mapping unit is limited. On the other hand, the performance of Check-In is greatly improved when the mapping unit is 4096 bytes, but the space overhead is increased by almost 3 % over ISC-C. In the key-value store, which mainly contains small data, the space overhead can increase if the mapping unit becomes large. Also, the number of invalid pages is too large compared to the data being written, which can shorten the lifetime. Therefore, appropriate trade-offs are required when selecting a mapping unit.

V. Related Work

Conventional systems have been optimized to address the overhead issues of data journaling and checkpointing. We present related work on journaling and checkpointing mechanisms and discusses recent changes in the key-value store and SSD architecture.

Previous journaling approaches. To reduce the overhead of data journaling, metadata journaling [9], compressed journaling [20], [53], and storage caching [16], [40], [41], [57] have emerged. However, these mechanisms do not fully take advantage of flash memory characteristics.

Transactional SSDs. Transactional SSDs such as TxFlash [52], Atomic-Write [50], and LightTx [46] support transactional interfaces to eliminate log stacking issues [56] that overlap log management of the DBMS or file system and log management of the FTL. Accessing these SSDs with page-level atomic commands often result in random I/O operations because they access the final location directly. Inside the SSDs, the number of commands that need to be processed increases, and their own logging is required to guarantee the atomicity of these commands. If most database applications have the performance bottleneck issue by the storage device, dedicating transaction roles to the storage device can aggravate the overall performance of the system.

Key-value SSDs. In recent years, there have been various researches on the SSDs for key-value stores, such as KAML [32], KVSSD [55], and Crail-KV [12]. Unlike traditional transactional SSDs, these key-value SSDs manage data with keys and values within the SSD which can reduce the overhead of converting and maintaining pages at an upper layer. However, these key-value SSD approaches require a lot of resources. Especially, much more memory resources are required in the storage device rather than the main memory. In addition, if the server system uses storage devices with a special interface, the whole system needs to be modified. Also, storage engines with reduced roles can lose their advantages.

Flash-aware key-value stores. There are many studies on key-value stores that recognize the characteristics of flash memory [21], [22], [42]–[44], [47]. They use a block interface and propose the data structure to reduce the write amplification of flash memory. Compared to our work, data consistency works need to by the host CPU, increasing the remapping overhead, and they have low remapping efficiency due to insufficient utilization of the key-value on FTL.

Advanced FTL designs. Similar to our research, there are content-aware FTLs [19], [27], [35], [45], [54], in-storage processing techniques [30], [39], and in-storage data deduplication techniques [38]. Based on the advantages of them, the proposed FTL can cooperate with the storage engine. Our FTL uses a mapping method to fit the various record sizes supported in the storage engine. It also provides operational flexibility by offloaded functions from the storage engine.

Near data processing for checkpointing. Recently, techniques for offloading checkpointing to the storage device level have been studied [7], [24], [28], [36] but they are less effective in key-value stores with variable data sizes. High-Performance Checkpoint/Restart [7] offloads checkpointing into the SSD to reduce the CPU overhead but it does not solve the redundant writes problem on NVM. WAL-SSD [28] also proposed a remapping method for journal data to reduce redundant writes to storage devices like our approach. However, Check-In, which supports sector-aligned journaling, improves remapping efficiency and it is more effective in key-value stores with variable data sizes by reducing invalid page generation than WAL-SSD. Likewise, previous SSD checkpointing schemes [24], [36] have a similar concept of offloading checkpointing, but they are not suitable for variable data format in key-value stores.

VI. Conclusion

We proposed the Check-In mechanism to reduce the checkpointing overhead. The FTL which recognizes checkpointing in an SSD can prevent unnecessary data duplication due to checkpointing. The proposed sector-aligned journaling and remapping method prevents redundant flash operations and extends the lifetime of flash memory. Shortened checkpointing time greatly reduces the tail latency and provides a fairer service. It also supports recovery from power off or system crash in the device, like transactional SSDs. The benefits of processing data beyond sector size are obvious, but smaller data processing still needs to be optimized. Increased mapping information and reduced space utilization are challenges. Further research is required to optimize and realize the data consistency of the storage device in commercial key-value store systems.

Acknowledgment

This work was supported by the Memory Division of Samsung Electronics Co., Ltd and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2018R1A2A2A05018941). We also thank Professor Myoungsoo Jung for his valuable feedbacks. W. W. Ro is the corresponding author.

References

[1] "DBrank," https://db-engines.com/en/ranking_categories/.

[2] "LevelDB," https://github.com/google/leveldb.

[3] "MongoDB," https://docs.mongodb.com/manual/.

[4] "NVM express," http://www.nvmexpress.org/.

[5] "RocksDB," http://rocksdb.org.

[6] "YCSB," https://github.com/brianfrankcooper/YCSB.

[7] A. Agrawal, G. H. Loh, and J. Tuck, "Leveraging near data processing for high-performance checkpoint/restart," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*, 2017.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference (ATC'08)*, 2008.

[9] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Operating systems: Three easy pieces." Arpaci-Dusseau Books, 2018.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, 2011.

[12] T. Bisson, K. Chen, C. Choi, V. Balakrishnan, and Y. suk Kee, "Crail-KV: A high-performance distributed key-value store leveraging native KV-SSDs over NVMe-oF," in *IEEE 37th International Performance Computing and Communications Conference (IPCCC 2018)*, 2018.

[13] L. Bouganim, B. þór Jónsson, and P. Bonnet, "uFLIP: Understanding flash IO patterns," in *Proceedings of the Conference on Innovative Data Systems Research*, 2009.

[14] A. M. Caulfield, J. Coburn, T. I. Mollov, A. De, and A. Akel, "Understanding the impact of emerging non-volatile memories on high-performance in IO-intensive computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010.

[15] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of data (SIGMOD'18)*, 2018.

[16] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *The 32nd International Conference on Massive Storage Systems and Technology (MSST 2016)*, 2016.

[17] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS*, 2009.

[18] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in highspeed data processing," in *Proceeding of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[19] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[20] T.-Y. Chen, Y.-H. Chang, S.-H. Chen, C.-C. Kuo, M.-C. Yang, H.-W. Wei, and W.-K. Shih, "Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM," in *Design Automation Conference (DAC)*, 2017.

[21] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," in *Proceedings of the VLDB Endowment*, 2010.

[22] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD'11)*, 2011.

[23] 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 SIGOPS Operating Systems Review*, 2007.

[24] J. Do, D. Lomet, and I. L. Picoli, "Improving CPU I/O performance via SSD controller FTL support for batched writes," in *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*, 2019.

[25] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*, 2018.

[26] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, 2009.

[27] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[28] K. Han, H. Kim, and D. Shin, "WAL-SSD: Address remapping-based write-ahead-logging solid-state disks," in *IEEE Transactinos on Computers*, 2020.

[29] T. Harter, D. Borthakur, S. Dong, A. Aiyer, and L. Tang, "Analysis of HDFS under HBase: A Facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, 2014.

[30] W. S. Jeong, C. Lee, K. Kim, M. K. Yoon, W. Jeon, M. Jung, and W. W. Ro, "REACT: Scalable and high-performance regular expression pattern matching accelerator for in-storage processing," in *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[31] S. Jin, J. Kim, J. Kim, J. Huh, and S. Maeng, "Sector log: fine-grained storage management for solid state drives," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[32] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible and high-performance key-value SSD," in *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA 2017)*, 2017.

[33] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "SimpleSSD: modeling solid state drives for holistic system simulation," in *IEEE Computer Architecture Letters*, 2018.

[34] M. Kang, W. Lee, and S. Kim, "Subpage-aware solid state drive for improving lifetime and performance," in *IEEE Transactions on Computers*, 2018.

[35] Y. Kang, R. Pitchumani, T. Marlette, and E. L. Miller, "Muninn: a versioning flash key-value store using an object-based storage model," in *Proceedings of the 12th ACM Internatinoal Systems and Storage Conference (SYSTOR 2014)*, 2014.

[36] H. A. Khouzani and C. Yang, "Towards a scalable and write-free multi-version checkpointing scheme in solid state drives," in *Proceeding of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[37] D. Kim and S. Kang, "Partial page buffering for consumer devices with flash storage," in *2013 IEEE Third International Conference on Consumer Electronics Berlin (ICCE-Berlin)*, 2013.

[38] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. ung Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in SSDs - model and quantitative

analysis," in *The 28th International Conference on Massive Storage Systems and Technology (MSST 2012)*, 2012.

[39] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, 2017.

[40] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[41] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," in *IEEE Transactions on Computers*, 2015.

[42] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," in *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, 2013.

[43] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient and high-performance key-value store," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[44] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016.

[45] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[46] Y. Lu, J. Shuy, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD '13)*, 2013.

[47] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable and lightweight and FTL-aware key-value store," in *USENIX Annual Technical Conference (ATC'15)*, 2015.

[48] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[49] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL database: New era of databases for big data analytics—classification, characteristics and comparison," in *International Journal of Database Theory and Application (IJDTA)*, 2013.

[50] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proceedings of the 17th IEEE Symposium on High Performance Computer Architecture (HPCA 2011)*, 2011.

[51] Padhy, R. P., M. R. Patra, and S. C. Satapathy., "RDBMS to NoSQL: Reviewing some next-generation non-relational database's," in *International Journal of Advanced Engineering Science and Technologies*, 2011.

[52] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[53] D. J. Shoff, J. Liu, and J. C. Southmayd, "Compressed file system for non-volatile RAM," in *US Patent 6944742*, 2005.

[54] G. Wu and X. He, "Delta-FTL: Improving ssd lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys'12)*, 2012.

[55] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of lsm trees and flash translation layer for write-efficient KV store," in *Design and Automation and Test in Europe Conference and Exhibition (DATE 2018)*, 2018.

[56] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *The 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW '14).*, 2014.

[57] Z. Zhang, L. Ju, and Z. Jia, "Unified DRAM and NVM hybrid buffer cache architecture for reducing journaling overhead," in *Design and Automation Test in Europe Conference Exhibition (DATE)*, 2016.