Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

MDB-KCP: persistence framework of in-memory database with CRIU-based container checkpoint in Kubernetes

Jeongmin Lee[1], Hyeongbin Kang[1], Hyeon-jin Yu[1], Ji-Hyun Na[1], Jungbin Kim[1], Jae-hyuck Shin[1] and Seo-Young Noh[1*]

Abstract

As the demand for container technology and platforms increases due to the efficiency of IT resources, various workloads are being containerized. Although there are efforts to integrate various workloads into Kubernetes, the most widely used container platform today, the nature of containers makes it challenging to support persistence for memory-centric workloads like in-memory databases. In this paper, we discuss the drawbacks of one of the persistence support methods used for in-memory databases in a Kubernetes environment, namely, the data snapshot. To address these issues, we propose a compromise solution of using container checkpoints. Through this approach, we can perform checkpointing without incurring additional memory usage due to CoW, which is a problem in fork-based data snapshots during snapshot creation. Additionally, container checkpointing induces up to 7.1 times less downtime compared to the main process-based data snapshot. Furthermore, during database recovery, it is possible to achieve up to 11.3 times faster recovery compared to the data snapshot method.

Keywords Container, Kubernetes, In-memory database, Checkpoint/restore

Introduction

In the IT industry, enhancing the efficiency of computing resources is recognized as a key goal. The choice of computing technology to achieve this has become a major concern within the industry [1]. Traditionally, efforts were focused on improving resource efficiency through virtualization technology. However, virtualization faced performance degradation compared to bare-metal machines due to issues such as hardware virtualization, the computational overhead of additional virtualization software, and increased complexity [2].

Container technology has emerged as an alternative to address these challenges. Container technology provides advantages such as lower performance degradation compared to virtualization technology, thanks to environment isolation, while still improving resource efficiency [3]. Furthermore, container technology, based on features like environment isolation, lightweight design, and high scalability, is increasingly recognized as an excellent choice for achieving resource efficiency goals.

Currently, many IT services operate in container-based environments, realizing efficient and flexible management. Additionally, with the advancements in container technology, various solutions for efficient container management have emerged [4].

Kubernetes stands out as the most widely utilized platform today, serving as a container orchestration tool that automates management tasks such as deployment and scaling of containerized applications [5]. It has become a pivotal platform for effectively overseeing container-based operational environments. Kubernetes offers features like lifecycle management, automatic

*Correspondence:
Seo-Young Noh
rsyoung@cbnu.ac.kr
[1] Department of Computer Science, Chungbuk National University, Cheongju 28644, South Korea

recovery, rolling updates, and rollback for containers, going beyond simple container management to enhance the high availability and fault tolerance of containerized IT services.

Consequently, there is a growing trend towards integrating diverse workloads into Kubernetes-based environments. Particularly, there is a demand for containerizing and integrating stateful workloads, including database applications, data processing applications, and monolithic legacy services reliant on stateful configurations [4].

For managing stateful workloads in a containerized environment, Kubernetes provides a feature called PersistentVolume (PV). This feature allows containers to maintain persistence by storing the data they use in storage volumes. However, this approach does not capture the container's internal state, including context, process execution, library loading, cache, and memory state. In essence, while PV is valuable for workloads that require persistence by storing data in storage, it may not be effective for workloads that demand persistence while managing data in memory.

One prominent workload with such characteristics is the in-memory database. Unlike disk-based databases that store the primary data on disk and cache only a portion of the critical data in memory, in-memory databases store the entire primary data in memory. As the cost of memory continues to decline and the input/output speed of hard disks struggles to keep up with the performance of other computing elements, in-memory databases that can manage and provide data at a faster pace are gaining attention [6]. Additionally, there is a growing use of in-memory databases in IT service operations, driven by the increasing demand for big data and real-time processing applications [7].

The drawback of in-memory databases lies in their volatile nature. In the event of a database server shutdown or restart, the data present in the database is lost. Given the nature of in-memory databases, maintaining persistence is challenging. While some in-memory database products partially support persistence by storing snapshots of data at specific points in time on disk, the process of restoring these snapshots from disk to memory can be time-consuming. Furthermore, there are methods of executing queries based on logs, but querying all the data stored in the database results in significant time consumption. In other words, extended downtime during failure recovery is a significant implication, leading to availability problem. Moreover, the conventional fork-based data snapshot method employed by in-memory databases poses an issue wherein additional memory, up to twice the data size, is utilized when write operations are initiated on the instance.

Since in-memory databases are primarily sought after for applications requiring large volumes of data and rapid data processing, extended downtime during database failure and recovery processes can compromise the reliability of applications. Therefore, efforts to minimize downtime during failure recovery in in-memory databases are crucial. Moreover, amidst the trend of containerization and integration into Kubernetes environments for numerous workloads, it is essential to explore the advantages of container-based operations for in-memory databases operating in such environments.

In this paper, we propose and validate the use of container snapshots based on CRIU (Checkpoint/Restore in Userspace) [8], a feature supported by Container Runtime Interface – Open Container Initiative (CRI-O) [9], one of the container runtime interfaces in Kubernetes, as a method for maintaining persistence in in-memory databases within the Kubernetes environment. We aim to compare the traditional approach of loading data snapshots in in-memory databases with the method of deploying container snapshots directly as containers in Kubernetes. Our goal is to discuss the advantages and applicability of these approaches. This study contributes to the following aspects:

- Verification of the issues with the traditional fork-based data snapshot approach in a Kubernetes environment.
- Reduction of downtime in the checkpoint process of in-memory databases by using container checkpoints compared to the main process-based data snapshot approach.
- Decreased recovery time compared to the data snapshot approach by leveraging container checkpoints in case of database failures.

The rest of this paper is organized as follows. First, we present the background on the key technologies in "Background" section. Next, "Related work" section describe works related to the persistence of in-memory databases and studies concerning the CRIU-based container checkpoint technology. "Problem statement" section discusses the challenges associated with the conventional data snapshot approach for maintaining persistence in in-memory databases. In the "Proposed method" section, we introduce the container checkpoint approach as an alternative to the data snapshot method and elaborate on its features and advantages. Following that, performance evaluation is given in "Experiments and performance analysis" section. Last, "Conclusion" section concludes this paper and proposes future work.

Background

In this section, we describe the characteristics of the methods used to maintain the persistence of in-memory databases and introduce the CRIU technology underlying the container checkpoint methodology.

Persistence in in-memory database

With the emergence of workloads demanding big data and real-time processing, rapid data processing and analysis have become crucial challenges. However, traditional disk-based database systems faced difficulties in guaranteeing fast response times due to disk I/O being a primary performance bottleneck [10]. To address this, in-memory database systems have been introduced. These systems store data in memory rather than on disk, providing faster data access compared to disk-based databases. However, the drawback of in-memory databases is that the volatile medium, memory, does not support persistence. Fundamentally, databases are required to support ACID properties, but achieving durability in in-memory databases, where data is stored in memory, is challenging. In other words, in the event of a database failure where workload interruption or restart is inevitable, all data may be lost. Therefore, many in-memory database products use the following methods during database failure recovery to ensure persistence [11, 12]:

- **Data Snapshot**: Periodically copying all data stored in memory into a snapshot form on non-volatile disk, and during database failure, reloading the snapshot from disk to memory after workload recovery. However, this method has the issue of losing changes made after snapshot creation.
- **Transaction Logging**: Storing a log file of all insert and update operations performed by the database on non-volatile disk, and during database failure, executing all queries recorded in the log file to recover data. The drawback of this approach is that recovery takes a considerable amount of time since the database needs to execute all queries.

In practical scenarios of in-memory database failure recovery, a combination of the above methods is often used to offset their respective drawbacks. Typically, a data recovery through snapshots is performed initially, and for changes made after snapshot creation, the transaction logging method is employed [13].

Another challenge in in-memory database failure recovery is the data snapshot creation process. There are various algorithms for creating data snapshots in in-memory databases, and among them, Redis, the most widely used in the market, employs two snapshot creation methods., each with its own problems:

- **Main Process-based Data Snapshot Creation**: Blocking the operation of the in-memory database, saving a snapshot file of the current memory state to disk. During the snapshot creation process, the database blocks read/write commands requested by clients, leading to downtime in the workload.
- **Fork-based Data Snapshot Creation**: Utilizing child process forked from the main process for snapshot creation. Although this method avoids downtime during the snapshot creation process, it may lead to a memory usage problem, as the database performs copy-on-write when changes occur during the snapshot creation process, potentially doubling the memory usage. Running out of memory resources can cause the database service to crash.

Figure 1 depicts the process of main process-based data snapshot creation and fork-based data snapshot creation. The most significant difference between the two approaches lies in the occurrence of downtime in the database service. In the main process-based data snapshot method, the main process, responsible for read/write operations in the database, directly performs the snapshot creation. As a result, the database service is unavailable during the snapshot creation period. In contrast, the fork-based data snapshot creation method utilizes a *fork()* system call to create a child process that performs the snapshot creation [14]. Consequently, the database service remains available even during the snapshot creation period.

Given the downtime issue during snapshot creation in the main process method and the potential failure due to excessive memory usage in the child process creation method, creating snapshots directly on a database server where actual operations occur is deemed unstable. Therefore, databases are structured in a Active-Standby configuration, where the Active instance provides actual services, and the Standby instance mirrors the data from the Active while performing snapshot creation [15, 16].

Figure 2 illustrates the workflow for data snapshot creation to maintain persistence when a write operation occurs in the Active-Standby structure of an in-memory database. When changes occur due to a write operation on the Active instance, if the Standby instance is successfully connected, the write operation is directly executed on the Standby instance as well. However, if the connection to the Standby instance is disrupted due to downtime, the write operation is temporarily stored in the Backlog Buffer of the Active. Data stored in the Backlog Buffer has a replication offset, allowing for partial

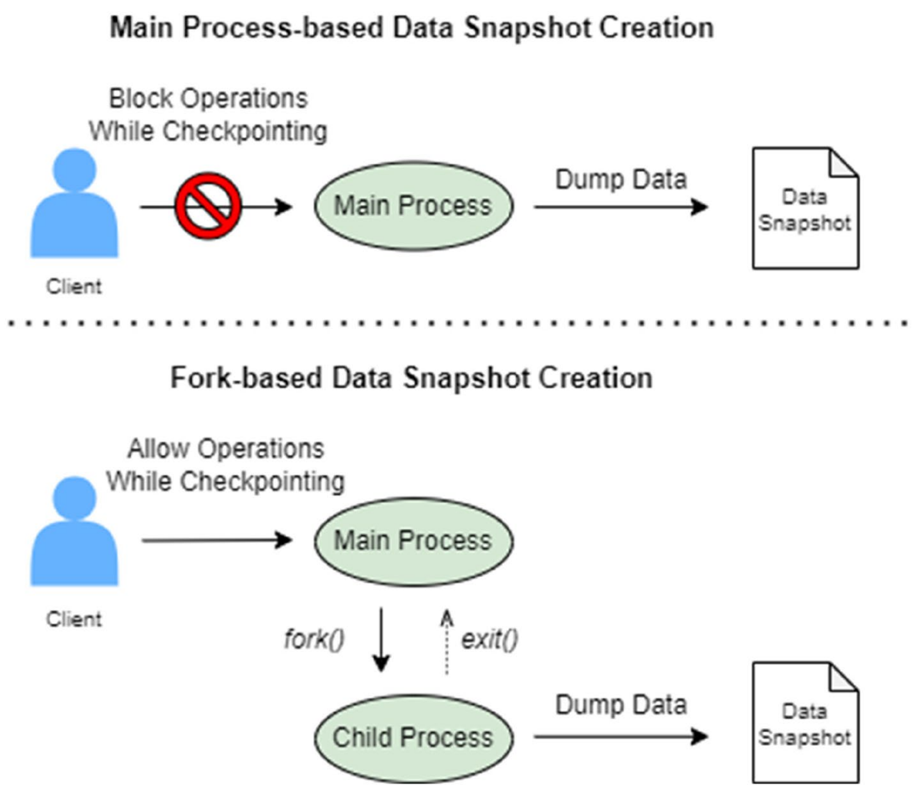
Main Process-based Data Snapshot Creation
Block Operations While Checkpointing
Main Process
Dump Data
Data Snapshot
Client
Fork-based Data Snapshot Creation
Allow Operations While Checkpointing
Main Process
Client
fork()
exit()
Child Process
Dump Data
Data Snapshot

Fig. 1 Main process-based data snapshot creation and fork-based data snapshot creation

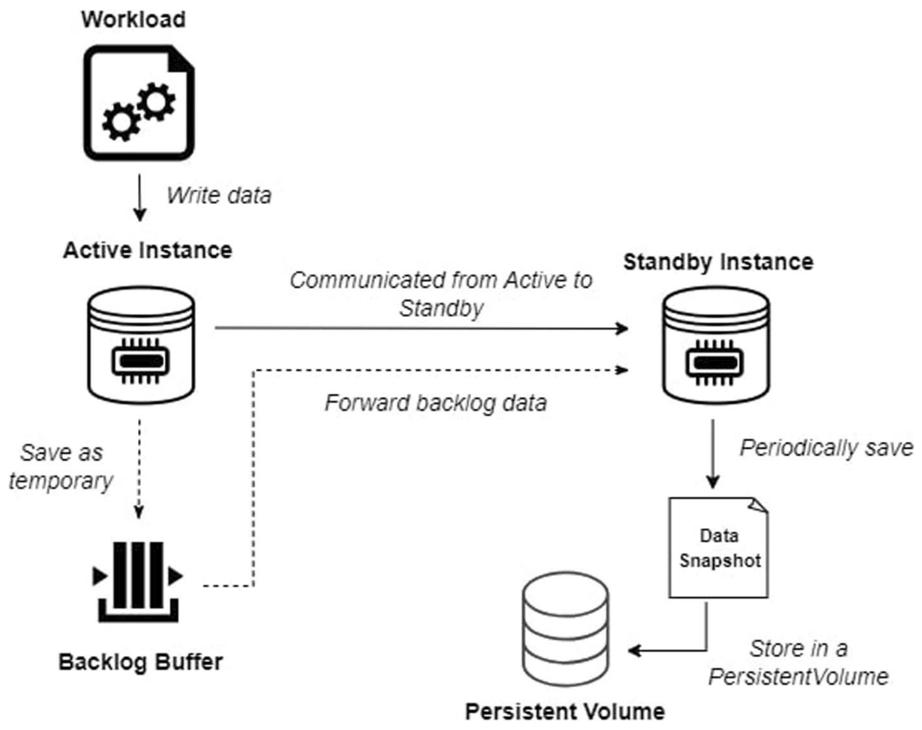

Fig. 2 Data snapshot creation workflow in active-standby structure

synchronization by forwarding the data from the buffer when a normal connection is reestablished with the Standby instance, without relying on snapshot-file-based data synchronization.

Checkpoint / restore in userspace (CRIU)

CRIU is a Linux software that supports checkpoint/restore functionality for Linux processes through memory dumps. It saves the current state of a process to disk and performs restoration based on the stored archive.

CRIU's checkpoint process involves maintaining the consistency of processes by fetching the Process ID (PID) of the target process, recursively collecting process information (i.e. files descriptors, pipe parameters, memory maps), and freezing it. During this checkpoint, it is crucial that the process being dumped remains transparent, and any changes during the process state transition should not be noticed. Hence, utilizing the *ptrace()* interface, it transparently captures and freezes the process, employing parasitic code injection techniques to obtain the state of the target process without killing it [17].

Related work

In this section, we discuss research related to the persistence of in-memory databases and studies concerning the CRIU-based container checkpoint technology.

Firstly, there have been several studies focusing on maintaining the persistence of in-memory databases [18] conducted a performance evaluation and comparison of the traditional fork-based data snapshot method and the mainstream snapshot algorithm commonly used for maintaining persistence in existing in-memory database products. The study observed that the fork-based data snapshot method performs better in update-intensive workloads [19] proposed a checkpointing system using Validity Tracking Compression (VTC), a technique that tracks the validity of logs, to address the problem of doubling memory usage in update-intensive workloads caused by fork-based snapshots. This system ensures that only an additional 2% of memory is required during checkpointing.

There are also several studies on CRIU-based container checkpointing technology.

Bhardwaj et al. [20] addresses challenges such as system failures and load balancing in fog computing environments through the utilization of LXD container virtualization technology and CRIU-based migration. Tran et al. [21] proposes a framework for migrating containerized services using CRIU technology in Kubernetes environments. In this study, Redis, an in-memory database product, was also used as one of the benchmark targets.

Our research stands out from previous studies by focusing on using CRIU for container checkpoint and recovery specifically tailored to in-memory database workloads. In particular, our research distinguishes itself by leveraging container checkpoints, especially concerning the persistence aspect of in-memory databases. This paper analyzes the differences between the persistence mechanisms inherent to in-memory database workloads and the utilization of container checkpoints for maintaining persistence. It explores the advantages of using container checkpoints in achieving persistence.

Problem statement

In this section, we discuss the drawbacks of the conventional data snapshot approach used for maintaining persistence in in-memory databases. Specifically, we elaborate on the issues related to additional memory usage in the fork-based data snapshot method, which allows snapshot creation without downtime, and experimentally verify whether such problems arise in a Kubernetes environment.

Problem with Fork-based data snapshots in Kubernetes

In the Fork-based data snapshot method, the main process forks a child process to perform the snapshot creation. Since the child process handles the snapshot creation, it does not impact the main process responsible for operating the database, allowing operations on the database. However, during the snapshot process, if there are modifications to the database instance, the main process copies the corresponding memory pages before making modifications. In the case of a write operation on the database triggering modifications, additional memory space is required to copy the memory pages. If modifications occur in all memory pages, it necessitates up to twice the maximum data size in memory space [18, 19]. If the additional memory required during the snapshot process is not allocated, an Out of Memory (OOM) issue may occur, leading to system failure. The above issues are the same when running container-based in-memory databases in a Kubernetes environment. This can be a concern in container environments where various workloads, beyond the database instance, operate on a single node. Furthermore, Kubernetes does not support Swap memory, so even if the node's memory resources are insufficient, memory space cannot be allocated from devices other than the main storage [22]. Therefore, in Kubernetes environments, the need for efficient utilization of memory resources is emphasized.

We conducted experiments to investigate additional memory usage during the execution of fork-based checkpointing in a Redis container, one of the most actively used in-memory databases in the IT market, running in a Kubernetes environment. Using the Yahoo! Cloud Serving Benchmark (YCSB) [23] tool, we triggered write operations on the database to measure the additional memory usage due to Copy-On-Write (CoW) compared to the size of the stored data in the database.

The testbed on which this experiment was run is the same as the testbed in the Experiments and performance analysis section.

Table 1 shows the parameter configurations for the YCSB workload used to benchmark the database in the experiment. To assess the occurrence of CoW in various workloads using the database, we adjusted the number of records and the update operation ratio. The size of each record stored in the database used the default record size provided by YCSB. Additionally, to maintain consistency in the number of commands generated in the database during the checkpoint process in each experiment, the thread count was standardized to 16.

Figure 3 presents the results of an experiment measuring the additional memory usage due to CoW during update operations with the YCSB benchmark during the checkpoint process of a container-based in-memory database performed in a Kubernetes environment. In Fig. 3a shows the additional memory usage due to CoW as a percentage of data size for a read-oriented workload with 10% update operations during the checkpoint. This shows up to 22% additional memory usage compared to the memory usage of the original Redis instance. Figure 3b shows the additional memory usage due to CoW in a workload balanced with 50% read and 50% update operations. It shows up to 55% additional memory usage compared to the memory usage of the original Redis instance. Figure 3c shows the memory usage incurred due to CoW in a write-oriented workload with a 90% update job ratio. It shows up to 70% additional memory usage compared to the memory usage of the original Redis instance. As a result, for all YCSB workload benchmarks, the additional memory usage due to CoW increases as the number of records increases, with more memory usage experienced in write-oriented workloads.

Fork-based checkpointing provides the advantage of not impacting database operations during the checkpoint process, making it a favorable solution for performing checkpoints in operational environments where database downtime is undesirable. However, in architectures like Active-Standby, where the operational instance and the instance performing the checkpoint are separate, even if downtime occurs in the checkpointing instance, it does not affect the database service itself. In such a structure, there is no reason to use fork-based checkpointing while tolerating inefficient memory usage during the checkpoint process. Therefore, an alternative checkpoint/restore solution that efficiently utilizes memory resources in an environment where the operational instance and checkpoint instance are separated is needed.

Table 1 YCSB tool configuration

Parameters	Values
Record Count	1M, 2M, 4M, 8M, 16M
Update Ratio	10%, 50%, 90%
Record Size	default
Number of Threads	16
Distribution	zipfian

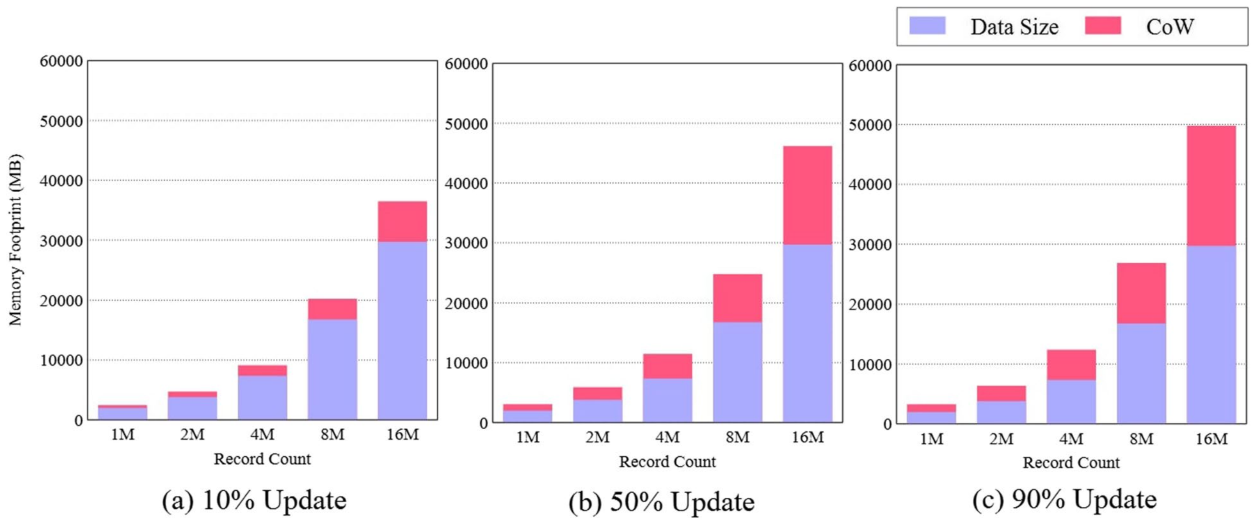

Fig. 3 CoW occurrence during the checkpointing with Redis container in Kubernetes

Loading time of data snapshot

The data snapshot method includes loading data from a stored snapshot file on the disk when a database needs to restart due to any unforeseen reason. In particular, when operating an in-memory database as a container on orchestration platforms like Kubernetes, if a failure occurs in the database, it triggers a restart of the container. In the event of a failure in the database container resulting in a restart, during the loading process, the database becomes temporarily inaccessible, becoming a major factor in increased downtime for recovery in in-memory databases.

For workloads where real-time performance is crucial, as often seen in in-memory databases with faster data input/output than disk-based databases, prolonged recovery times can compromise the reliability of the service. Therefore, a recovery solution is needed that can quickly restore services in the event of a failure, processed and prepared for fast recovery before the occurrence of a failure, rather than loading data at the time of failure. This ensures both real-time capabilities and service stability, minimizing the Recovery Time Objective (RTO).

Proposed method

Our goal is to perform database checkpointing without wasting memory resources due to CoW. The conventional methods involve checkpointing through the main process of the database. Checkpointing through the main process essentially blocks operations during the checkpoint process, rendering the database inaccessible. Therefore, the entire checkpointing process results in database downtime. In an independent structure where the operational instance and checkpointing instance are separate, the downtime of the checkpointing instance does not affect database operations. However, as the checkpoint time increases, the size of the backlog buffer used for synchronization between the Active and Standby instances after restoration grows. This leads to increased memory usage in the Active Instance and longer synchronization times. In cases where more data accumulates in the backlog buffer than the configured size, creating a snapshot file on the Active instance and loading the entire database into memory are required for synchronizing the Active and Standby instances.

Furthermore, we aim to provide fast database services by minimizing downtime during the restore process for efficient fault recovery. Traditional in-memory databases create snapshot files during the checkpoint process and go through the steps of restarting the instance after a database failure and loading the snapshot file. During the time it takes to load the snapshot file, the database is unavailable, resulting in downtime for that instance. Therefore, a solution is needed that focuses on quick recovery before the occurrence of a fault, rather than loading data at the time of the fault.

In-memory databases require periodic checkpointing to maintain persistence, and fast restoring is essential to ensure real-time responsiveness. Therefore, an efficient solution is needed to minimize instance downtime and restore time during the checkpointing process without wasting memory resources. We propose the use of CRIU for container-based in-memory databases operating in a Kubernetes environment as a solution to maintain persistence.

CRIU with Kubernetes environment

CRIU has been integrated with various container runtime platforms such as LXC, LXD, Docker, Podman, and more. Kubernetes, being one of the most widely used container runtime platforms in the container market, supports CRIU, providing container checkpoint and restore capabilities. Both the low-level container runtime interfaces in Kubernetes, namely *runc* and *crun*, and the high-level container runtime interface, CRI-O, integrate with CRIU, enabling the storage and restoration of container states. In traditional container runtime platforms, CRIU was primarily used for container migration purposes. Consequently, when executing a checkpoint command, the existing running container was stopped. For example, in Docker, when creating a container checkpoint using the *docker checkpoint create* command, the state of the existing running container transitions to the exited state [24].

However, CRIU with Kubernetes behaves differently. CRIU with Kubernetes is developed for the purpose of live container forensics [25]. When performing a container checkpoint, it can recognize that the container is being checkpointed and still proceed to store the container snapshot without deleting the actively running container. In other words, even when a checkpoint command is executed, the running container is not terminated and continues its operation.

Figure 4 illustrates the container checkpoint workflow using CRIU in a Kubernetes environment. Kubernetes CRIU triggers the container snapshot request to the *kubelet*, an agent process responsible for managing container execution, using URL-based approach. The URL specifies the container's namespace, pod, and container. Through this process, a checkpoint TAR archive for the specified container is generated on the disk. Subsequently, the container checkpoint TAR archive is transformed into the Open Container Initiative (OCI) image using an external container build tool. During this transformation, an annotation is added to the container image, indicating that the container has been checkpointed. Finally, the container is deployed into the Kubernetes

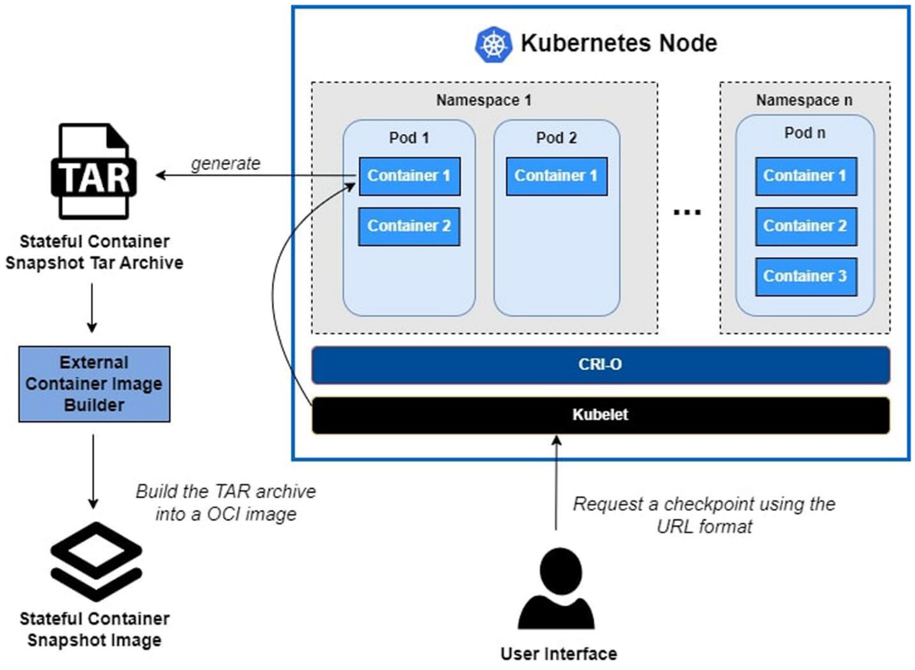
Kubernetes Node
Namespace 1
Pod 1
Container 1
Container 2
Pod 2
Container 1
Namespace n
Pod n
Container 1
Container 2
Container 3
CRI-O
Kubelet
generate
Stateful Container Snapshot Tar Archive
External Container Image Builder
Build the TAR archive into a OCI image
Stateful Container Snapshot Image
Request a checkpoint using the URL format
User Interface

Fig. 4 Container checkpoint workflow with CRIU in a Kubernetes environment

environment using the same deployment method as the existing Kubernetes container images.

In-memory database checkpoint with container snapshot

The checkpoint process is essential to ensure data persistence in in-memory databases. Traditional checkpoint methods, such as fork-based data snapshots, incur significant memory overhead due to the CoW mechanism. Additionally, main process-based data snapshots result in downtime for the entire checkpoint process. In contrast, the container checkpoint method leverages CRIU to capture the state of the running container, minimizing the impact on the operational database instance.

Checkpointing can be executed either on the operational instance or on a dedicated backup instance, with each approach having distinct impacts on downtime and memory usage. Operational Instance Checkpointing involves performing checkpoints directly on the active database instance. This method ensures continuous service availability as key database operations continue uninterrupted. However, due to the CoW mechanism, memory usage may increase during the checkpointing process. On the other hand, Backup Instance Checkpointing offloads the checkpointing process to a separate instance to mitigate its impact on the operational instance. This method requires robust synchronization to ensure that the backup instance accurately reflects the state of the operational instance. The main advantages are preventing additional memory usage and service downtime on the active instance. Checkpointing an in-memory database using CRIU-based container snapshots offers several advantages over traditional data snapshot methods:

Minimized Downtime: Traditional main process-based data snapshots result in significant downtime as the database service is entirely halted during the checkpoint process. In contrast, the container checkpoint method incurs downtime only during the memory dump and tar archive creation phases, significantly reducing service interruption compared to main process-based data snapshots.

Optimized Memory Usage: Fork-based snapshots can lead to substantial memory usage due to the CoW mechanism when write operations occur on the instance. The container checkpoint method prevents additional memory usage on the instance. This approach leverages Kubernetes' capability to efficiently manage containerized workloads, making the checkpoint process both effective and resource efficient.

Table 2 shows the distinctly differentiated characteristics between the traditional data snapshot method and the container checkpoint method for checkpointing in-memory databases.

By adopting the container checkpoint method, the checkpoint process for in-memory databases can

Table 2 Characteristics of data snapshot and container checkpoint when checkpointing in-memory database

Method	Downtime during Checkpoint	Additional Memory Usage	Complexity
Main process-based Data Snapshot	High	None	Low
Fork-based Data Snapshot	None	Up to 2 times (due to CoW)	Moderate
CRIU Container Checkpoint	Low	None	High

reduce downtime compared to the main process-based data snapshot method and decrease memory usage compared to the fork-based data snapshot method. This approach enhances overall performance and reliability in a Kubernetes environment, making it a suitable solution for modern containerized applications.

This structured approach ensures that in-memory databases maintain high availability and performance, leveraging Kubernetes' capabilities for efficient workload management and minimizing downtime and memory usage during the checkpoint process.

In-memory database recovery with container snapshot

In-memory databases are vulnerable to data loss during failures or restarts because they store primary data in memory. Unlike disk-based databases, which do not require data reloading during recovery, in-memory databases must reload data into memory during recovery, potentially causing downtime. The traditional recovery method involves loading data from snapshot files stored on disk, which can take time and affect service availability. The traditional recovery methods for in-memory databases, such as data snapshots, involve significant challenges related to data volatility and downtime:

> **Data Volatility**: In-memory databases store data in volatile memory, leading to potential data loss during restarts or failures. This necessitates reloading data from disk-based snapshot files, a process that can be time-consuming.
> **Downtime**: The process of loading data from snapshot files during recovery incurs substantial downtime. The database remains unavailable until the data loading process is complete, which can severely impact applications that require high availability and real-time performance.

This paper aims to address the challenges mentioned above by using CRIU container checkpoints to perform recovery of in-memory databases in case of failures. Recovery of in-memory databases using CRIU container snapshots offers significant advantages over traditional methods:

> **Preloaded Data**: Container snapshots capture the state of the data already loaded in memory at the time of the snapshot. This preloaded state allows for the omission of the data loading step during recovery, enabling faster restoration.
> **Reduced Downtime**: By deploying a new container using the pre-captured snapshot image, the data loading step is bypassed, significantly reducing downtime. This ensures that the database service is quickly restored and able to handle client requests promptly.

Figure 5 illustrates the timeline of the recovery scenario when a failure occurs in the in-memory database instance. In the conventional container-based in-memory database, recovery involves restarting the instance at the time of failure and loading data from the stored data snapshot file. However, with container snapshot-based in-memory database restoration, the process is performed by deploying a container based on the workload snapshot container image stored at the time of failure.

Therefore, the container checkpoint method pre-includes the data to be loaded into memory during the checkpoint process. While the container checkpoint may take longer for snapshot creation compared to the traditional data snapshot method, considering Recovery Point Objective (RPO), it offers an advantage in RTO by enabling faster service restoration. Table 3 shows the characteristics observed during the recovery of in-memory databases using traditional data snapshots compared to container checkpoints.

Checkpoint and restore with container checkpoint for practical use: MDB-KCP

The existing data snapshot method for maintaining the persistence of in-memory databases is a feature provided by the database application itself. This involves periodically creating snapshots and automatically loading the data when the database needs to restart. In contrast, the container checkpoint discussed in this paper is a manual process in the Kubernetes environment. Administrators need to send a checkpoint creation request directly to the *kubelet* using *curl*, and the process of converting the generated TAR archive into an OCI image also needs to be done manually through a container image builder. Furthermore, since the state-saved container image is separate from the default database container, it must be manually deployed by specifying it in a separate YAML file. Therefore, to automate container checkpoints similar to the data snapshot method, additional requirements need to be satisfied. To maintain the persistence of an in-memory database, the following functionalities are required:

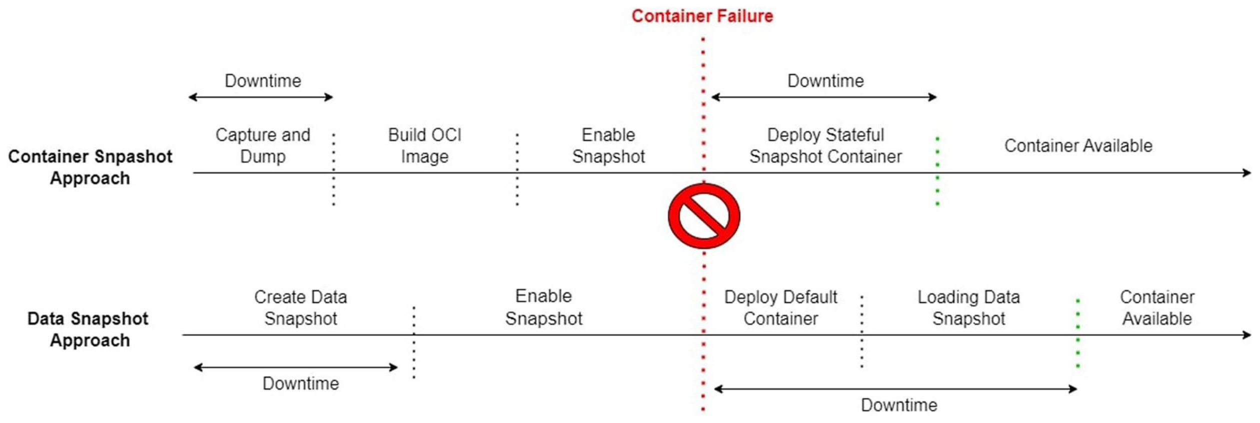

Fig. 5 In-memory database checkpoint and recovery timeline with container snapshot approach and data snapshot approach

- Periodic checkpointing of the database.
- Automatic detection of database failures and restoration to the checkpointed state.

In this section, we propose the MDB-KCP (Memory Database with Kubernetes Checkpoint) framework to meet the requirements for maintaining the persistence of in-memory databases using Kubernetes' CRIU container checkpoints [26]. First, the implementation of the Kubernetes CRIU container checkpoint method for in-memory databases involves several steps:

1. **Periodic Checkpoint Requests**: Initiate the checkpoint process for the target container by sending requests to the Kubernetes *kubelet*.
2. **Checkpoint Execution with CRIU**: The *kubelet* uses Checkpoint/Restore In Userspace (CRIU) to capture the container's state, including memory, process state, and file descriptors, and stores this state in a TAR archive.
3. **Conversion to OCI Image**: Convert the TAR archive into an Open Container Initiative (OCI) image using container image build tools, and store this OCI image for future restoration.
4. **Backup Instance Deployment**: Deploy the OCI image as a backup instance within the Kubernetes environment, ensuring that the backup instance accurately reflects the state of the operational instance through synchronization mechanisms.
5. **Monitoring and Synchronization**: Continuous monitoring and synchronization between the operational and backup instances are required. The synchronization mechanism ensures that the backup instance remains consistent with the operational instance.

Table 3 Characteristics of data snapshot and container checkpoint when restoring in-memory database

Method	Recovery Time from Failure	Service Downtime after Failure	Complexity
Data Snapshot	High	High	Low
CRIU Container Checkpoint	Low	Low	High

Algorithm 1 MDB-KCP: Checkpointer

```
Define: Checkpoint Period p, Kubelet agent K, OCI image builder B, URL pointing to an in-memory database container deployed in Kubernetes url, In-memory database state S
1: p = input()
2: procedure CHECKPOINT(url, p)
3:   while true do
4:     Send checkpoint request using curl with url to K
5:     path = K.criuDump(S)
       // Save state (memory, process state, file descriptors) in TAR archive and return the path of TAR archive
6:     B.buildImage(path)
       // Convert TAR archive to OCI image and save to localhost registry
7:     Wait for p seconds
8: end
```

Initially, the administrator specifies the frequency at which the database checkpoint should occur. MDB-KCP: Checkpointer then sends a checkpoint request in the *curl* format to the *kubelet* according to the specified interval. The *kubelet* captures and dumps the database container using the *criu dump* command, creating a stateful container TAR archive. Subsequently, MDB-KCP: Checkpointer builds the TAR archive into the OCI image format through the container image builder. This enables the periodic checkpointing of the database. Algorithm 1 shows the process by which MDB-KCP: Checkpointer performs checkpointing of an in-memory database container.

Monitoring whether failures occur in the container and, if a failure is detected, redeploying the container using the checkpointed state-saving OCI image are essential tasks. Kubernetes can automatically redeploy a container if a failure occurs, thanks to the liveness probe. However, for container recovery using a checkpoint image, a different automatic container deployment feature is required. This is because it involves deploying a separate container image that captures the state information of the actively running database, rather than the default container image used by the database. Implementing the container checkpoint recovery process involves these steps:

1. **Monitoring and Failure Detection**: Continuously monitor the state of the database container to promptly detect failures. Utilize Kubernetes' liveness probes to monitor the health status of the container.
2. **Initiating Recovery**: Upon detecting a failure, remove the failed container and deploy a new container using the pre-captured OCI image.
3. **Verifying Restoration**: After deployment, verify that the new container has been correctly restored. Ensure that the database is ready to handle client requests and that data integrity is maintained.
4. **Synchronization and Finalization**: Apply a synchronization mechanism to ensure that changes made after the snapshot creation are reflected in the restored instance. This process may involve replaying transaction logs or other methods.

To meet these requirements, we propose introducing MDB-KCP: Restorer to interact with existing components, providing an architecture that fulfills these functionalities.

Algorithm 2 MDB-KCP: Restorer

```
Define: Container state check period $p$, Kubelet agent $\mathbf{K}$,
Kubernetes API Server $\mathbf{S}$, In-memory database container
deployed in Kubernetes $C$, State-saved in-memory database
checkpoint $N$
1: A = new A() // Initialize client configure
2: $p$ = input()
3: procedure RESTORE($p$,$C$,$N$)
4:   while true do
5:     $C$.state = A.fetchContainerState($C$)
6:     if $C$.state == Error then
7:       A.deleteContainer($C$)
8:       A.deployContainer($N$)
9:     Wait for $p$ seconds
10: end
```

Algorithm 2 shows the interaction between the components required for the recovery of an in-memory database through container checkpointing in the form of a sequence diagram. Initially, MDB-KCP: Restorer must continuously monitor the state of the target database container through the Kubernetes API server. MDB-KCP: Restorer periodically pulls the container's state metrics from the Kubernetes API server. In the event of a failure in the database container, MDB-KCP: Restorer detects it, removes the failed database container, and deploys a new container using the specified Kubernetes YAML file containing the stateful OCI image for recovery. This enables the automatic recovery of in-memory database containers in the Kubernetes environment.

Using the two modules mentioned above, container checkpointing for maintaining the persistence of an in-memory database can be implemented easily with shell scripts or programming languages such as Python.

Experiments and performance analysis

In this section, we conduct experiments comparing the checkpoint and restoration processes of traditional in-memory databases using the data snapshot approach with those using CRIU-based container snapshots in the Kubernetes environment.

Testbed setup

We conducted all experiments by deploying in-memory database containers in a Kubernetes environment with a single node. The node is equipped with an Intel Xeon Silver 4208 CPU running at 2.10 GHz; the CPU had 8 physical cores and 16 logical cores with hyperthreading enabled. We use Dell 2TB 7.2K RPM SATA 6Gbps 512n hard drive for storing and loading data snapshot and container snapshot. The machine ran CentOS 9 distribution with kernel 5.14. Additionally, we configured a single-node Kubernetes 1.28.1 based on the CRI-O container runtime interface.

Application setup

The in-memory database application utilized Redis 7.2.1. Furthermore, the data snapshot protocol for maintaining the persistence of the in-memory database employed Redis's RDB. The data stored in the in-memory database was bulk-loaded with 10M, 20M, 40M, 80M, and 160M records using Redis's *Debug Populate* command.

Checkpointing time

First, we compared the data snapshot creation time in Kubernetes-based in-memory database containers with the creation time of stateful OCI images in container checkpoints. In the Redis in-memory database, the following data snapshot creation protocols exist:

- **RDB-BGSAVE**: A forked child process performs the data snapshot creation process in the background. During this time, the operation of the database is not interrupted.
- **RDB-SAVE**: The main process performs the data snapshot creation process. The operation of the database is paused during the snapshot creation time.

In this experiment, we compare the snapshot creation time and downtime incurred on the container for both protocols, **RDB-BGSAVE**, **RDB-SAVE**, and the container checkpoint approach. For container checkpoints, we measured the downtime-inducing tar archive creation process time and the OCI image build process time, which occurs in a separate process from the instance and does not incur downtime.

Figure 6 shows the time taken for main process-based and fork-based data snapshot creation, as well as container checkpointing for checkpointing an in-memory database container. Both the data snapshot approach and the container checkpoint approach show an increase in time proportional to the data size of the in-memory database. In terms of snapshot creation time, container checkpointing consumes up to 2 times more time compared to the data snapshot approach. Fork-based data snapshotting incurs no downtime for the container, while the entire checkpoint time in the main process-based data snapshot results in downtime for the in-memory database container. Container checkpointing incurs downtime only in the process that captures the container and generates a tar archive by dumping the memory. Therefore, in terms of downtime incurred on the container, the container checkpointing approach has up to 7.1 times less downtime compared to the data snapshot approach. As a result, container checkpointing consumes more time for snapshot creation compared to the data snapshot approach, but incurs less downtime on the container compared to the main process-based data snapshot approach.

Restoring time

Next, we compared the time required for each approach to restore data in the in-memory database. Although the processes for creating snapshots in **RDB-SAVE** and **RDB-BGSAVE**, the snapshot generation protocols used by Redis, differ, both protocols ultimately create data snapshots in the RDB format. Therefore, in this experiment, we compared the time required for database recovery using the loading of RDB format data snapshots and the deployment of container checkpoint images. The criteria for recovery completion were based on the database's ability to process incoming requests from clients.

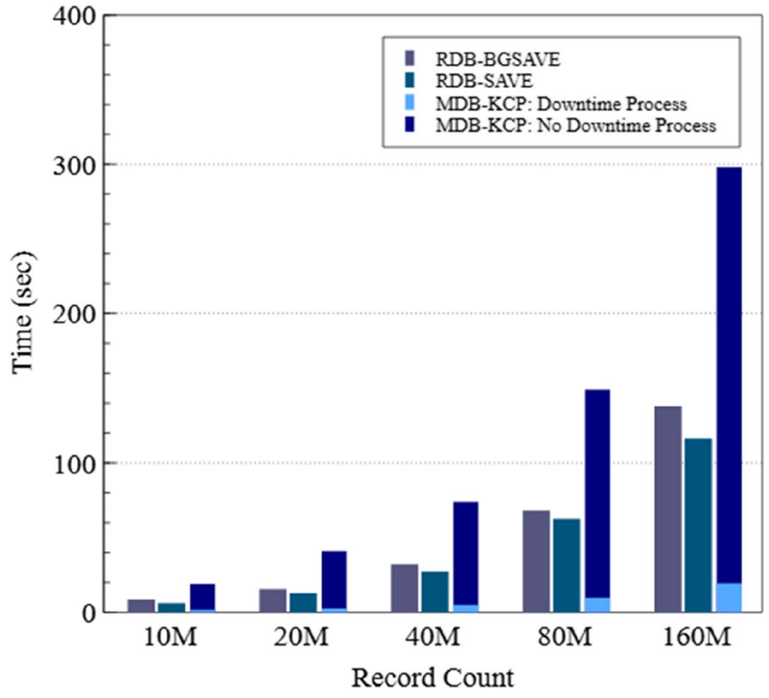

Fig. 6 Checkpointing time of data snapshot and container checkpoint

Figure 7 shows the data recovery time for the in-memory database container through data snapshots and container checkpoint snapshots. Experimental results show that recovery through container checkpoints is up to 11.3 times faster than the data snapshot recovery method. The recovery through the data snapshot method of the in-memory database involves most of the time during the process of deploying the base container and loading the data. The loading time is proportional to the size of the data stored in the in-memory database. On the other hand, in the database recovery based on container checkpoint, the data is already stored in the stateful container image during the checkpoint process. Therefore, during the container redeployment process, it is observed that relatively less time is required compared to the data snapshot method.

Conclusion

In this paper, we validate the drawbacks of the traditional fork-based data snapshot approach used for maintaining persistence in in-memory databases in a Kubernetes environment. As an alternative for ensuring persistence in in-memory databases, we propose the use of Kubernetes container checkpoints. To maintain the persistence of in-memory databases, the container checkpoint used has the following characteristics compared to the data snapshot approach:

- Low instance downtime compared to main process-based data snapshot
- Extended checkpoint snapshot creation time
- Short service recovery time

For an in-memory database operating in a structure where the operational environment and backup environment are separated, performing periodic checkpoints with relatively less downtime seems advantageous. Additionally, it enables checkpointing while preventing memory waste due to CoW. Furthermore, maintaining the persistence of in-memory databases through container checkpoints can offer the advantage of quick recovery speed, especially in workloads where real-time performance must be ensured.

However, in the Kubernetes environment, CRIU-based container checkpoints require conversion to the OCI image format, introducing the drawback of needing external container image build tools before deployment in Kubernetes. This process dominates most of the time in container checkpointing, resulting in container checkpoints taking up to twice as long as snapshot creation compared to the traditional data snapshot approach.

Therefore, in future research, we plan to explore container deployment approaches that can be managed in a Kubernetes environment without converting the state-saved container TAR archive to the OCI image format. This is expected to minimize the checkpoint snapshot creation time. Also, our experiments were conducted

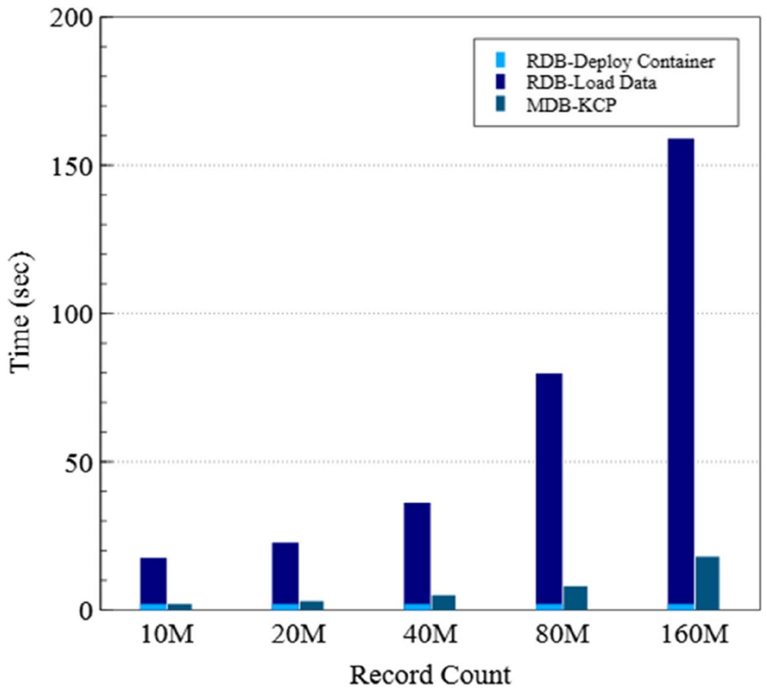

Fig. 7 Restoring time of data snapshot and container checkpoint

with Kubernetes operating in a single-node environment to simplify the scenario. In the future, we plan to investigate whether additional benefits exist by using container checkpoints for maintaining the persistence of in-memory databases in large-scale Kubernetes cluster environments.

Abbreviations

PV	Persistent Volume
CRIU	Checkpoint/Restore in Userspace
ACID	Atomicity, Consistency, Isolation, and Durability
OOM	Out-of-Memory
PID	Process ID
YCSB	Yahoo! Cloud Serving Benchmark
CoW	Copy-on-Write
RTO	Recovery Time Objective
RPO	Recovery Point Objective
OCI	Open Container Initiative

Acknowledgements
The authors would like to extend their sincere thanks to the Global Science Experimental Data Hub Center (GSDC) at the Korea Institute of Science Technology Information (KISTI) for their support of our research.

Authors' contributions
JL and HB designed the framework. JL drafted the manuscript. JL, HB, and JS contributed to the design of the experiments. JN and JK performed the experiments. SN supervised this study. All authors contributed to the final version of the manuscript and data analysis. The authors read and approved the final manuscript.

Funding
This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under grant number NRF-2008-00458, and by the Basic Science Research Program through the NRF funded by the Ministry of Education under grant number 2021R1I1A3053034.

Availability of data and materials
No datasets were generated or analysed during the current study.

Declarations

Competing interests
The authors declare no competing interests.

Received: 2 April 2024 Accepted: 9 July 2024
Published online: 25 July 2024

References
1. Beloglazov A, Buyya R, Lee YC, Zomaya A (2011) A taxonomy and survey of energy-efficient data centers and cloud computing systems. Adv Comput 82:47–111
2. Pelletingeas C (2010) Performance evaluation of virtualization with cloud computing (Doctoral dissertation)
3. Li Z, Kihl M, Lu Q, Andersson JA (2017) Performance overhead comparison between hypervisor and container based virtualization. In: 2017 IEEE 31st International Conference on advanced information networking and applications (AINA). Taipei, IEEE, p 955–962
4. CNCF (2020) CNCF SURVEY 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Accessed 14 Feb 2024
5. Rodriguez MA, Buyya R (2019) Container-based cluster orchestration systems: a taxonomy and future directions. Softw Pract Exp 49(5):698–719
6. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M et al (2008) Exascale computing study: technology challenges in achieving exascale systems. In: Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 15, 181
7. Abourezq M, Idrissi A (2016) Database-as-a-service for big data: an overview. Int J Adv Comput Sci Appl 7(1):157–177
8. CRIU. Checkpoint/Restore in Userspace. https://criu.org/Main_Page. Accessed 10 Dec 2023
9. cri-o. Lightweight container runtime for Kubernetetes. https://cri-o.io. Accessed 10 Feb 2024
10. Zhang H, Chen G, Ooi BC, Tan KL, Zhang M (2015) In-memory big data management and processing: a survey. IEEE Trans Knowl Data Eng 27(7):1920–1948
11. Magalhaes A, Monteiro JM, Brayner A (2021) Main memory database recovery: a survey. ACM Comput Surv (CSUR) 54(2):1–36
12. Bao X, Liu L, Xiao N, Lu Y, Cao W (2016) Persistence and recovery for in-memory NoSQL services: a measurement study. In: 2016 IEEE International Conference on Web Services (ICWS). San Francisco, IEEE, p 530–537
13. Redis. Redis persistence, how Redis writes data to disk. https://redis.io/docs/management/persistence. Accessed 3 Jan 2024
14. Park J, Lee Y, Yeom HY, Son Y (2020) Memory efficient fork-based checkpointing mechanism for in-memory database systems. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. p 420–427
15. EDUCBA (2023) Redis persistence. https://www.educba.com/redis-persistence. Accessed 10 Jan 2024
16. Shrestha R (2017) High availability and performance of database in the cloud-traditional master-slave replication versus modern cluster-based solutions
17. CRIU. Checkpoint/Restore. https://criu.org/Checkpoint/Restore. Accessed 10 Dec 2023
18. Li L, Wang G, Wu G, Yuan Y (2018) Consistent snapshot algorithms for in-memory database systems: experiments and analysis. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). Paris, IEEE, p 1284–1287
19. Lee K, Kim H, Yeom HY (2021) Validity tracking based log management for in-memory databases. IEEE Access 9:111493–111504
20. Bhardwaj A, Gupta U, Budhiraja I, Chaudhary R (2023) Container-based migration technique for fog computing architecture. In: 2023 international conference for advancement in technology (ICONAT). Goa, India, IEEE, p 1–6
21. Tran MN, Vu XT, Kim Y (2022) Proactive stateful fault-tolerant system for kubernetes containerized services. IEEE Access 10:102181–102194
22. Kubernetes (2023) Installing kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm. Accessed 10 Dec 2023
23. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. p 143–154
24. docker docs. docker checkpoint. https://docs.docker.com/engine/reference/commandline/checkpoint. Accessed 8 Jan 2024
25. Reber A (2022) Forensic container checkpointing in Kubernetes. https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha. Accessed 11 Nov 2023
26. MDB-KCP. Available: https://github.com/CBNU-DCLab/MDB-KCP. Accessed 20 May 2024